AFATL-TR-88-156

Image Algebra FORTRAN Preprocessor User's Manual

J N Wilson
D C Wilson
G X Ritter

UNIVERSITY OF FLORIDA
DEPARTMENT OF COMPUTER & INFORMATION SCIENCES
301 COMPUTER SCIENCE AND ENGINEERING BUILDING
GAINESVILLE, FLORIDA 32611

MARCH 1989

FINAL REPORT FOR PERIOD JANUARY TO DECEMBER 1987

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AIR FORCE ARMAMENT LABORATORY
Air Force Systems Command ▮ United States Air Force ▮ Eglin Air Force Base, Florida

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The AFATL STINFO program manager has reviewed this report, and it is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

Nicholas C. Hablenko

NICHOLAS C. HABLENKO, Lt Col, USAF
Acting Chief, Advanced Guidance Division

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify AFATL/ AGS , Eglin AFB FL 32542-5434.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFATL-TR-88-156
6a. NAME OF PERFORMING ORGANIZATION University of Florida Dept of Computer & Info Sciences 6b. OFFICE SYMBOL (If applicable) CIS	7a. NAME OF MONITORING ORGANIZATION Air-to-Surface Guidance Branch Advanced Guidance Division
6c. ADDRESS (City, State, and ZIP Code) 301 Computer Science and Engineering Building University of Florida Gainesville, Florida 32611	7b. ADDRESS (City, State, and ZIP Code) Air Force Armament Laboratory Eglin Air Force Base, Florida 32542-5434
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFATL/AGS & DARPA/TTO 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F08635-84-C-0295
8c. ADDRESS (City, State, and ZIP Code) AFATL/AGS, Eglin AFB, FL 32542-5434 DARPA/TTO, 1400 Wilson Blvd, Arlington, VA	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 62602F; PROJECT NO. 2068; TASK NO 06; WORK UNIT ACCESSION NO. 44

11. TITLE (Include Security Classification)

Image Algebra FORTRAN Preprocessor - User's Manual

12. PERSONAL AUTHOR(S)
Joseph N. Wilson, D. Clay Wilson, G. X. Ritter

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Jan 87 TO Dec 87	March 1989	85

16. SUPPLEMENTARY NOTATION

Availability of this report is specified on verso of front cover.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Image Algebra; Pattern Recognition; Mathematical Morphology; Image Segmentation; Image Processing
16	04		
17	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report provides an algorithm developer with a user's manual for the Image Algebra, FORTRAN 77, Preprocessor. The manual contains a thorough description of the capabilities and limitations of the preprocessor's code. Its emphasis is on providing a variety of step-by-step descriptions, with illustrations, that clearly demonstrate how to use the preprocessor syntax to express image processing transformations and information extraction techniques.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Patrick C. Coffield	22b TELEPHONE (Include Area Code) (904) 882-2838 22c OFFICE SYMBOL AFATL/AGS

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PREFACE

This report covers the period from June 1987 to December 1987. It was prepared by the Department of Computer and Information Sciences, 301 Computer Science and Engineering Building, University of Florida, Gainesville, Florida 32611, under Air Force Contract F08635-84-C-0295 with the Air Force Armament Laboratory, Eglin AFB, FL 32542-5434.

The authors wish to thank the Air Force Armament Laboratory (AFATL) and the Defense Advanced Research Projects Agency (DARPA) for sponsoring this work. We are particularly grateful to Dr. Sam Lambert (AFATL), Dr. Donald Daniel (AFATL), and to Dr. Jasper Lupo (DARPA) for sensing the importance of a solidly-grounded, usable image processing algebra, and for generously sharing their resources with us. We wish to acknowledge in a special way the support received from our program manager, Mr. Neal Urquhart (ASE). His constant encouragement, helpful comments, and untiring support made the work of this project more fruitful than anyone had thought possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION I
INTRODUCTION

This document describes the Image Algebra FORTRAN language and its preprocessor implementation at the University of Florida Center for Computer Vision Research. The Image Algebra is a comprehensive notation in which all image processing transformations can be expressed. The Image Algebra FORTRAN (IAF) preprocessor supports a subset of the Image Algebra, extending FORTRAN77 to support Image Algebra operations.

While this document is not intended to be a comprehensive introduction to the Image Algebra, an overview is presented of those Image Algebra constructs supported in IAF in the next section. The third section describes the extensions to FORTRAN77 provided by IAF. Example IAF programs showing some simple image processing transformations carried out in IAF programs are given. The document concludes with a guide explaining how to install Image Algebra FORTRAN on a given system.

SECTION II
A BRIEF TOUR OF THE IMAGE ALGEBRA

The Image Algebra is an algebra designed especially for the manipulation of images. This section introduces the Image Algebra by defining its operations and operands. This section briefly introduces only those Image Algebra operations and operands supported by the Image Algebra FORTRAN preprocessor. A complete introduction to the full Image Algebra can be found in Reference 1.

Any algebra consists of a set of operands and operations. In the subset of the Image Algebra supported by Image Algebra FORTRAN, the operands of interest are values, images, and templates. The Image Algebra supports unary and binary image operations that yield image results, unary and binary image operations that yield scalar valued results, and binary operations computing an image from an image and a template. This section defines and discusses each of the supported operands and operations in this section.

1. COORDINATE SETS, VALUE SETS, AND IMAGES

An *image* in the Image Algebra is a function from a coordinate set into a value set; that is, an image assigns some value to each coordinate (or pixel location) in some coordinate set.

Figure 1. Example Image

Figure 1 shows an image of an sr71 airplane. The values assigned to the coordinates of this image are integers in the range 0 to 31. Each value is assigned a shade of gray, white being assigned to value 0 and black being assigned to value 31. Such an image is represented in Image Algebra FORTRAN by an array. This representation of image as arrays means that the coordinate set of an image is merely the set of valid array subscripts for the array representing that image. So in Image Algebra FORTRAN all images are rectangular. Throughout this manual **a**, **b**, and other bold lowercase Roman letters at the beginning of the alphabet are used to represent images; **X** and **Y** are coordinate sets; and **x** and **y** with or without subscripts represent elements of coordinate sets.

2. UNARY AND BINARY IMAGE OPERATIONS

Any unary or binary operation defined over a set of gray values can be applied to images with that gray value set. If a binary operation is applied to two image arguments, those image arguments must have the same coordinate set. When applying an operation to image operands, the operation is applied pointwise to its arguments, yielding a result image with the same coordinate set as the arguments.

Figure 2. Images **a** and **b**

Consider the images **a** and **b** shown in Figure 2. Both of these images contain only pixels with values 0 and 1. Now consider what happens if one adds the images together. To do this we add the pixel values in each location of these two images to yield a result image **c**. The formal definition of addition of **a** and **b**, two images with coordinate set **X**, is given as follows:

$$\mathbf{a}+\mathbf{b} = \mathbf{c} \text{ where } \mathbf{c}(\mathbf{x}) = \mathbf{a}(\mathbf{x})+\mathbf{b}(\mathbf{x}) \text{ for all } \mathbf{x}\in\mathbf{X}.$$

Similarly, consider $\mathbf{d} = \mathbf{a}*\mathbf{b}$

$$\mathbf{a}*\mathbf{b} = \mathbf{d} \text{ where } \mathbf{d}(\mathbf{x}) = \mathbf{a}(\mathbf{x})*\mathbf{b}(\mathbf{x}) \text{ for all } \mathbf{x}\in\mathbf{X}.$$

The images of Figure 3 show the result of both adding and multiplying images **a** and **b** in this manner. In these images the value 0 is represented by white, value 1 by gray, and value 2 by black.

Figure 3. Images $\mathbf{c} = \mathbf{a}+\mathbf{b}$ and $\mathbf{d} = \mathbf{a}*\mathbf{b}$

In the Image Algebra, any binary or unary operation that can be performed on a gray value can be performed on an image in this fashion.

In addition to the usual FORTRAN operations, the Image Algebra supports other binary operations. The Image Algebra FORTRAN preprocessor represents these non-FORTRAN operations with mnemonic strings beginning with an exclamation point. In this treatment we use these mnemonics rather than Image Algebra Standard symbols. A table of mnemonic to symbol correspondence is given at the end of this section. The binary pointwise maximum and minimum are provided in the algebra. The maximum **e** of images **a** and **b** is defined to be

$$\mathbf{a}\ \texttt{!max}\ \mathbf{b} = \mathbf{e}, \text{ where } \mathbf{e}(\mathbf{x}) = \mathbf{max}(\mathrm{a}(\mathbf{x}),\mathrm{b}(\mathbf{x})) \text{ for all } \mathbf{x}\in\mathbf{X}.$$

Figure 4. Image $\mathbf{e} = \mathbf{a}\ \texttt{!max}\ \mathbf{b}$

The Image Algebra also supports an image dot product operation. The Image Algebra FORTRAN mnemonic operation symbol for the dot product is `!dot`. This operation is defined as

$$\mathbf{a}\ \texttt{!dot}\ \mathbf{b} = \sum_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x})*\mathbf{b}(\mathbf{x}).$$

When applied to an image consisting of only ones and zeros, this operation yields the number of pixels with value one.

Several new unary operations mapping an image into a scalar are provided as well: image sum, maximum, and minimum, represented respectively by the mnemonics `!sum`, `!max`, `!min`. The image sum yields the sum of the values at each pixel in the argument image, unary maximum yields the maximum value at any pixel, and minimum yields the minimum value at any pixel. That is, given **a** an image on **X**,

$$\texttt{!sum}\ \mathbf{a} = \sum_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x}), \quad \texttt{!max}\ \mathbf{a} = \bigvee_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x}), \quad \text{and } \texttt{!min}\ \mathbf{a} = \bigwedge_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x}).$$

Image thresholding is supported in the Image Algebra through the use of characteristic functions. A characteristic function is a binary operation that takes two image arguments

and gives as its result a new image with each pixel having value zero or one. In each of the characteristic functions defined below, a comparison is made. If the specified comparison is satisfied at a given pixel location, the result pixel value is one, otherwise the resulting pixel will have value zero.

Figure 5. Image **a**

Consider the image **a** shown in Figure 5, having values from 0 to 31; and the image **b** that has value 5 everywhere. Then given the characteristic function > defined as follows

$$(\mathbf{a}>\mathbf{b}) = \mathbf{c}, \text{ where } \mathbf{c} = \{(\mathbf{x},\mathbf{c}(\mathbf{x})):\mathbf{c}(x) = 1 \text{ if } \mathbf{a}(\mathbf{x}) > \mathbf{b}(\mathbf{x}), \text{ and 0 otherwise}\}$$

We normally represent a constant image having value k everywhere by the constant k. Hence we could write the characteristic expression above as **a**>5. Figure 6 shows the result of **a**>5 and **a**>30.

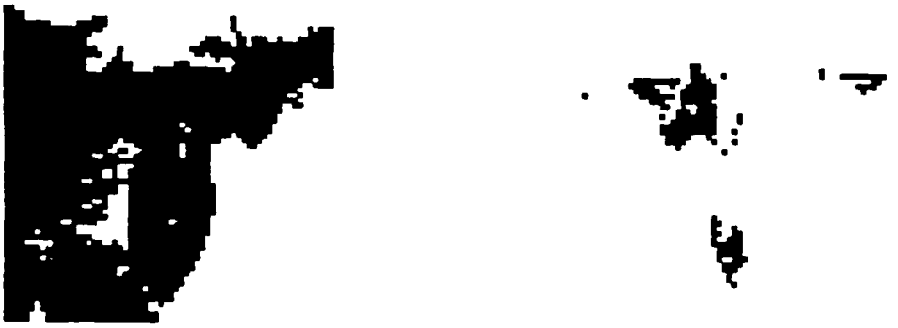

Figure 6. **a**>5 and **a**>30

Other characteristic functions supported are ==, yielding a one where its left argument and right argument are equal; !=, yielding one where its arguments are not equal; and >=, <, and <= yielding ones where the left argument is greater or equal, less, and less or equal, respectively.

3. TEMPLATES AND TEMPLATE OPERATIONS

Image Algebra templates and template operations permit one to use arithmetic/logic combinations of groups of pixel values in a source image to compute new values in a result image. Local neighborhood operations like local averaging represent a subset of the possible template operations.

An Image Algebra template is a function assigning to each coordinate in a result image over coordinate set $\mathbf{Y}$, an image over a source coordinate set $\mathbf{X}$. We call such a function a template from $\mathbf{Y}$ into $\mathbf{X}$. If a template $\mathbf{t}$ is a template from $\mathbf{Y}$ into $\mathbf{X}$, then we denote the image over $\mathbf{X}$ assigned to particular location $\mathbf{y}$ $\mathbf{t}_\mathbf{y}$.

Given an image $\mathbf{t}_\mathbf{y}$, some pixels may have the value zero. In practice, most of the pixel values are zero in an image defined by a template. The non-zero portion of the image assigned $\mathbf{t}_\mathbf{y}$ is called the *configuration* of the template at $\mathbf{y}$. The configuration of a template image $\mathbf{t}_\mathbf{y}$, also called the support of the image, is denoted $\mathcal{S}(\mathbf{t}_\mathbf{y})$. This is shown in the case of a template assigning an image on coordinate set $\mathbf{X}$ to each coordinate in a image over coordinate set $\mathbf{Y}$. This is shown pictorially in Figure 7.

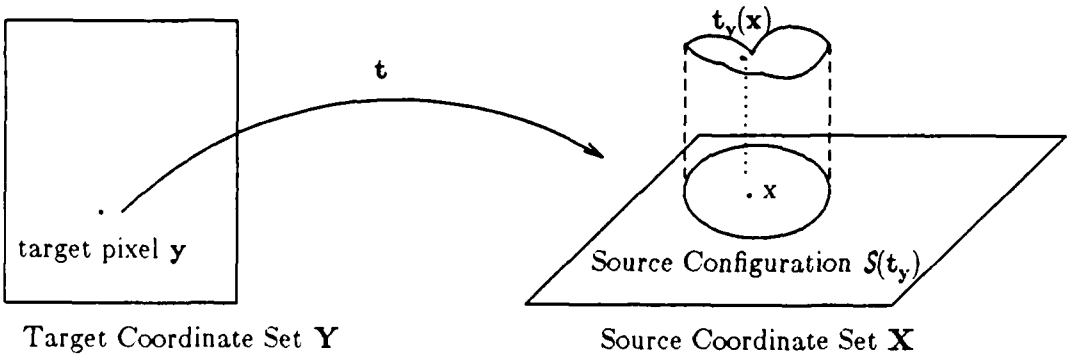

Figure 7. A pictorial example of a template from $\mathbf{Y}$ to $\mathbf{X}$.

A special class of templates are said to be *translation invariant*. A template is translation invariant whenever $\mathbf{t}_\mathbf{y}(\mathbf{x}) = \mathbf{t}_{\mathbf{y}+\mathbf{z}}(\mathbf{x}+\mathbf{z})$. Stated in words, a template is translation invariant whenever given any two coordinates $\mathbf{y}_1$ and $\mathbf{y}_2$, where $\mathbf{y}_2 = \mathbf{y}_1+\mathbf{z}$, the translation through vector $\mathbf{z}$ of the image $\mathbf{t}_{\mathbf{y}_1}$ is identically the image $\mathbf{t}_{\mathbf{y}_2}$. Such a template can be described by a single image that can be translated in the source coordinate set to describe every other target coordinate's configuration.

When a template is translation invariant, we draw its configuration by representing pixel locations in the configuration as boxes containing weights and denoting the target pixel location by shading the corners of its box. For example, given a two dimensional coordinate set and an arbitrary coordinate point $\mathbf{y} = (y_1, y_2)$, then Figure 8 represents the template $\mathbf{t}$, where

$$\begin{aligned} \mathbf{t}_{(y_1,y_2)}((y_1, y_2)) &= 1 \\ \mathbf{t}_{(y_1,y_2)}((y_1-1, y_2)) &= 2 \\ \mathbf{t}_{(y_1,y_2)}((y_1, y_2+1)) &= 3 \\ \mathbf{t}_{(y_1,y_2)}((y_1+1, y_2)) &= 4 \\ \mathbf{t}_{(y_1,y_2)}((y_1, y_2-1)) &= 5 \end{aligned}$$

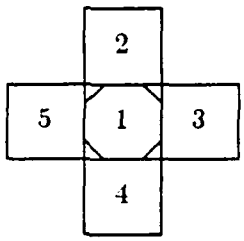

Figure 8. A translation invariant template

Note that in the above example we use matrix relative coordinate positioning. That is, the first coordinate of a coordinate pair describes row position and increases downward; the second coordinate describes column position and increases to the right. We use this convention throughout the document and in the Image Algebra FORTRAN preprocessor.

There are three basic template operations in the Image Algebra: *generalized convolution* (with mnemonic `!gcon`), multiplicative maximum (`!mmax`), *multiplicative minimum* (`!mmin`), *additive maximum* (`!amax`), and *additive minimum* (`!amin`). Generalized convolution takes as its arguments an image and a template. Let $\mathbf{t}$ be a template from $\mathbf{Y}$ to $\mathbf{X}$ and $\mathbf{a}$ an image on $\mathbf{X}$. The convolution of image $\mathbf{a}$ with template $\mathbf{t}$ is defined to be

$$\mathbf{a} \text{ !gcon } \mathbf{t} = \mathbf{c}, \text{ where } \mathbf{c}(\mathbf{y}) = \sum_{x \in X} \mathbf{a}(\mathbf{x}) * \mathbf{t}_{\mathbf{y}}(\mathbf{x})$$

This definition says that in the image $\mathbf{c}$, any pixel $\mathbf{y}$ gets its value by taking the image $\mathbf{t}_{\mathbf{y}}$, multiplying it by the image $\mathbf{a}$, and then summing all the pixel values. Notice that at points $\mathbf{x}$ outside the configuration of $\mathbf{t}_{\mathbf{y}}$ the values of $\mathbf{t}_{\mathbf{y}}(\mathbf{x})$ are all zero, so if we let $S(\mathbf{t}_{\mathbf{y}})$ denote the configuration of $\mathbf{t}_{\mathbf{y}}$, then we can also define this template operation as

$$\mathbf{a} \text{ !gcon } \mathbf{t} = \mathbf{c}, \text{ where } \mathbf{c}(\mathbf{y}) = \sum_{\mathbf{x} \in S(\mathbf{t}_\mathbf{y})} \mathbf{a}(\mathbf{x}) * \mathbf{t}_\mathbf{y}(\mathbf{x})$$

To describe the application of template operations, we apply the generalized convolution operation to the image of Figure 5 and the translation invariant template of Figure 9. This template will weight each pixel in the cross shaped neighborhood around a pixel by value 1 5. When the template operation is performed, the resultant pixels will have values representing neighborhood or local averages of the surrounding pixel values. The result is shown in Figure 10.

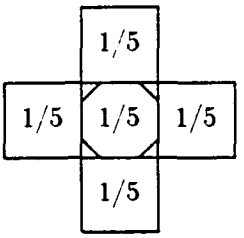

Figure 9. Averaging template $\mathbf{t}_{\mathbf{avg}}$

Figure 10. Image **a** and the result of **a** !**gcon** $\mathbf{t}_{\mathbf{avg}}$

Translation variant templates are those in which the configurations and weights at a given point are not rigidly translated from point to point. Consider the problem of shrinking an image, that is, making it fit into a smaller coordinate set. Suppose we know that image **a** of Figure 5 is defined on the coordinate set $\mathbf{X} = \left\{(\mathbf{x}_1, \mathbf{x}_2) : \mathbf{x}_1, \mathbf{x}_2 \in \{0, \cdots, 63\}\right\}$ and we wish to shrink this image by averaging neighboring pixels to fit into the coordinate set

$\mathbf{Y} = \left\{(\mathbf{y}_1, \mathbf{y}_2) : \mathbf{y}_1, \mathbf{y}_2 \in \{0, \cdots, 31\}\right\}$. We might use a template $\mathbf{s}$ defined at point $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$ as follows:

$$\mathbf{s}_y(2*\mathbf{y}_1, 2*\mathbf{y}_2) = 1/4,$$
$$\mathbf{s}_y(2*\mathbf{y}_1+1, 2*\mathbf{y}_2) = 1/4,$$
$$\mathbf{s}_y(2*\mathbf{y}_1, 2*\mathbf{y}_2+1) = 1/4, \text{ and}$$
$$\mathbf{s}_y(2*\mathbf{y}_1+1, 2*\mathbf{y}_2+1) = 1/4.$$

The result of this template operation is shown in Figure 11. Note that it is impossible to draw a single picture characterizing the relationship between the target pixel and its configuration in **s**. To illustrate this, consider the configurations for target coordinates (5, 5) and (6, 6). For target pixel (5, 5), the configuration is the set {(10,10), (10,11), (11,10), (11,11)}, whereas target pixel (6,6) has configuration, {(12,12), (12,13), (13,12), (13,13)}. The configuration for (6,6) is **not** a translation of the configuration (5,5) through vector (6,6)-(5,5).

Figure 11. **a!gcons**

The other image template operations use maximum and minimum operations. Just as zero is the identity with respect to addition, $-\infty$ is the identity with respect to maximum and ∞ is the identity with respect to minimum. Hence instead of defining the configuration of a template with respect to that portion of the image not containing the value zero, we define the configurations with the function $S_{\infty}(\mathbf{t}_y)$ which represents the coordinate set on which $\mathbf{t}_y$ does not take on either the value ∞ or $-\infty$. Formally defined, $S_{\infty}(\mathbf{t}_y) = \left\{\mathbf{x} : \mathbf{t}_y(\mathbf{x}) \notin \{\infty, -\infty\}\right\}$. In practice, since the values ∞ and $-\infty$ cannot be represented numerically, the Image Algebra FORTRAN preprocessor never actually stores these values. It only stores the values of pixels within the configuration of a template. With

this in mind, we define the rest of the image template operations as follows:

$$\mathbf{a}\,\texttt{!mmaxt} = \mathbf{c}, \text{ where } \mathbf{c(y)} = \bigvee_{x \in S_{\infty d}(t_y)} \mathbf{a(x)} * \mathbf{t_y(x)}$$

$$\mathbf{a}\,\texttt{!mmint} = \mathbf{c}, \text{ where } \mathbf{c(y)} = \bigwedge_{x \in S_{\infty d}(t_y)} \mathbf{a(x)} * \mathbf{t_y(x)}$$

$$\mathbf{a}\,\texttt{!amaxt} = \mathbf{c}, \text{ where } \mathbf{c(y)} = \bigvee_{x \in S_{\infty d}(t_y)} \mathbf{a(x)} + \mathbf{t_y(x)}$$

$$\mathbf{a}\,\texttt{!amint} = \mathbf{c}, \text{ where } \mathbf{c(y)} = \bigwedge_{x \in S_{\infty d}(t_y)} \mathbf{a(x)} + \mathbf{t_y(x)}$$

Section III of this document shows several complete Image Algebra FORTRAN programs and discusses some other template definitions and their uses.

4. CORRESPONDENCE OF SYMBOLS TO MNEMONICS

The following tables show the correspondence between standard Image Algebra operation symbols and Image Algebra FORTRAN mnemonic symbols.

Table 1. UNARY OPERATIONS SYMBOLS AND MNEMONICS

Operation Name	Standard Image Algebra Symbol	IAF Operation Symbol
Negation	$-$	-
Sum	Σ	!sum
Maximum	$\vee$	!max

Table 2. BINARY OPERATION SYMBOLS AND MNEMONICS

Operation Name	Standard Image Algebra Symbol	IAF Operation Symbol
Addition	+	+
Multiplication	*	*
Maximum	$\vee$	!max
Minimum	$\wedge$	!min
Exponentiation	**	**
Dot Product	$\bullet$	!dot

Table 3. CHARACTERISTIC FUNCTION SYMBOLS AND MNEMONICS

Characteristic Function	IAF Representation
$\chi_{=_b}{}^a$	a == b
$\chi_{\neq_b}{}^a$	a != b
$\chi_{>_b}{}^a$	a > b
$\chi_{\geq_b}{}^a$	a >= b
$\chi_{<_b}{}^a$	a < b
$\chi_{\leq_b}{}^a$	a <= b

Table 4. IMAGE TEMPLATE OPERATION SYMBOLS AND MNEMONICS

Operation Name	Standard Image Algebra Symbol	IAF Operation Symbol
generalized convolution	$\oplus$	!gcon
additive maximum	$\boxplus$	!amax
additive minimum	$\boxminus$	!amin
multiplicative maximum	$\circledvee$	!mmax
multiplicative minimum	$\circledwedge$	!mmin

SECTION III
IMAGE ALGEBRA EXTENSIONS TO FORTRAN 77

Image Algebra FORTRAN (IAF) is an extension of the ANSI Standard FORTRAN 77 programming language (ANSI X3.9-1978). More specifically, the IAF language is a composite of ANSI Standard FORTRAN 77 and a significant subset of the Image Algebra. The lexical, syntactic, and semantic aspects of IAF that support the Image Algebra extensions are defined in this section. Concrete examples of the use of IAF are given in the next section. It is assumed that the reader is familiar with both FORTRAN 77 and the Image Algebra.

The authors have implemented a preprocessor for IAF that translates IAF source code into semantically equivalent FORTRAN 77 code. This manual describes that implementation of IAF. The definition of IAF presented here is intended to coincide as closely as possible to the current preprocessor implementation. Any apparent discrepancy between the behavior of the preprocessor and the meaning ascribed to IAF in this document should be reported to the authors. The preprocessor was written in strict conformance to ANSI Standard FORTRAN 77 making the tool particularly suitable for single-language environments as well as making it easily portable.

1. OVERVIEW

Two principal features distinguish Image Algebra FORTRAN from FORTRAN 77. First, IAF supports the relatively unconstrained use of *images* as operands in arbitrary arithmetic expressions and as targets of arithmetic assignments. Indeed, any programming language that purports to support image processing applications must allow for images to be used as indivisible entities. In IAF, an image is simply a programmer-defined FORTRAN 77 array. Second, IAF supports Image Algebra *templates* which can be used to effect a virtually limitless variety of image transformations. IAF templates are defined by the programmer in a new kind of program unit designed specifically for that purpose. The *definition* of a template is syntactically very similar to the definition of a `FUNCTION` in a FORTRAN 77 program. Similarly, a *reference* to a template in an IAF arithmetic expression is syntactically identical to a reference to a `FUNCTION` or simple variable in a FORTRAN 77 expression. The IAF language is most notably differentiated from the FORTRAN 77 language by (i) its considerably richer arithmetic expressions, (ii) a more general arithmetic assignment statement, and (iii) the support of Image Algebra templates. The IAF language is described in

order of the following topics: (i) arithmetic operands, (ii) arithmetic operators, (iii) templates and template operators, (iv) IAF expressions and their use, and (v) exceptions and restrictions.

2. IMAGE ALGEBRA FORTRAN OPERANDS

In FORTRAN 77, a *primary* in an arithmetic expression can be one of the following: (i) an unsigned arithmetic constant, (ii) a symbolic name of an arithmetic constant (i.e., defined in a **PARAMETER** statement), (iii) an arithmetic variable reference, (iv) an arithmetic array element reference, (v) an arithmetic function reference, or (vi) an arithmetic expression enclosed in parentheses. Furthermore, in FORTRAN 77 all arithmetic primaries represent scalar values and all arithmetic expressions produce scalar results. In IAF, the unsubscripted name of a user-defined array can also be used as a primary in an arithmetic expression. The appearance of an unsubscripted array name within an arithmetic expression denotes its use as an *image*. Thus, an arithmetic primary in an IAF arithmetic expression may represent either a scalar value *or* an image value. Furthermore, IAF arithmetic expressions can produce scalar results or image results.

The terms *array* and *image* are used throughout this manual. To avoid possible confusion, a consistent use of these two terms is applied as follows. The term *array* is used in reference to a FORTRAN 77 array that does not explicitly involve any of the Image Algebra extensions of IAF. The term *image* is used explicitly in the context of the Image Algebra extensions. Note that the term array has meaning in the context of both FORTRAN 77 and IAF whereas the term image has meaning only in the context of IAF.

a. Arrays as Images

Images are the prototypical operands of IAF expressions. A user-defined IAF image is simply a FORTRAN 77 array to which certain restrictions apply. An IAF image is declared with exactly the same syntax that is used for declaring a FORTRAN 77 array. An IAF image is referenced in an arithmetic expression by using the unsubscripted name of the image as a primary in an arithmetic expression. In addition, the unsubscripted name of an IAF image may appear alone on the left-hand side of the '=' symbol in an arithmetic assignment statement.

Not all FORTRAN 77 arrays are valid for use as IAF images. The following restrictions are imposed on IAF images.

- The valid data types for IAF images are **INTEGER**, **REAL**, **DOUBLE PRECISION**, and **COMPLEX**.
- Every IAF image must be declared with exactly two-dimensions.
- All expressions that declare the dimension bounds in the declaration of an image must be constant **INTEGER** expressions.

The last restriction dictates that the size of all images must be statically determined at compile time. Thus, the use of either an adjustable array (i.e., an array declared with a dimension bound expression that references an **INTEGER** variable that is a dummy argument) or an assumed-size array (i.e., an array declared with '*' as the upper bound of the last dimension) as an IAF image is prohibited.

An array that is declared in violation of one or more of the above restrictions can not be used as an IAF image in any context. However, such an array may be used freely within an IAF program in any manner that conforms with the rules governing the use of arrays in FORTRAN 77.

The explicit use of an array as an image within an IAF program unit by using the unsubscripted name of the array either as a primary in an arithmetic expression or as the target of an arithmetic assignment statement does not preclude the use of that same array in any manner consistent with FORTRAN 77 in the same IAF expression or in any other arithmetic expression in the same program unit.

In compliance with the rules of FORTRAN 77, the declaration of the dimensions and bounds of an image may be given in a type-statement, a **DIMENSION** statement, or a **COMMON** statement. The type of an IAF image is determined by exactly the same rules that determine the type of a FORTRAN 77 array.

In the following fragment of IAF code, all of the array declarations are valid declarations for IAF images.

```
INTEGER A(64, 64), B(64, 64), CHAIN(1, 32)
PARAMETER (NROWS=64, NCOLS=64)
COMPLEX CIMAGE(NROWS, NCOLS)
DIMENSION DIMAGE(512, 512)
DOUBLE PRECISION D(0:NROWS/2-1, 0:NCOLS*2-1), DIMAGE
COMMON /COM/ RIMAGE(1024, 1024)
```

In particular, the declaration of **CHAIN** above illustrates the point that a two-dimensional array that is declared such that one of the dimensions has a size of 1 is a valid IAF image.

In contrast to the preceding array declarations, none of the arrays declared in the next fragment of IAF code are valid IAF images. However, they are valid IAF array declarations because they are valid FORTRAN 77 array declarations.

```
SUBROUTINE SUBR (W, X, M, N)
REAL V(32, 32, 32, 32, 32, 32, 32)
INTEGER W(M*2, N*2)
COMPLEX X(512, *)
INTEGER Y(256)
LOGICAL Z(64, 64)
```

In summary, all valid FORTRAN 77 arrays are also valid IAF arrays. However, only a subset of the valid FORTRAN 77 arrays are valid IAF images.

(1) Image Coordinate Sets

Every IAF image possesses a property known as its *coordinate set*. The coordinate set of a user-defined image is the dimension bounds of the underlying array. Thus, the coordinate set of a user-defined image is obtained directly from the array declarator used to declare the image. Every operation involving images possesses a *manifest* coordinate set. The manifest coordinate set of an operation involving one or more image operands plays a key role in defining and verifying the semantics of the operation. In particular, the manifest coordinate set of a given image operation is used as described below.

- If the underlying operation involves two or more image operands, then the manifest coordinate set of the operation is used to verify that the several images involved have *compatible* coordinate-sets.
- If the underlying operation dictates that an image is produced as a result, then the manifest coordinate set determines the coordinate set of the result image.

The operations involving images for which manifest coordinate sets are relevant are categorized listed below.

(1) Unary arithmetic operations in which the operand is an image or an arithmetic expression that produces an image result.

(2) Binary arithmetic operations (excluding image-template operations) in which at least one of the two operands is an image or an arithmetic expression that produces an image result.

(3) Image-template operations.

(4) Arithmetic assignment operations in which the left-hand side is a user-defined image.

(5) Arithmetic function references having at least one actual argument that (i) is an image or an arithmetic expression that produces an image result and, (ii) is prepended with the '@' symbol.

(6) Arithmetic array element references having at least one indexing expression that (i) is an image or an arithmetic expression that produces an image result and, (ii) is prepended with the '@' symbol.

The semantic issues concerning image coordinate sets in general, and manifest coordinate sets in particular, are discussed as the various IAF operations are described.

b. Scalars as Images

In certain contexts within IAF arithmetic expressions and arithmetic assignment statements, a scalar value is interpreted as an image. In cases where such an interpretation arises, the image is referred to as an implicit image. It is emphasized that any arbitrary expression that produces a scalar result is eligible for interpretation as an implicit image. It is the specific context in which a scalar expression occurs which dictates whether or not it denotes an implicit image.

There are two ways in which a scalar value can assume the identity of an implicit image. The most common situation in which this occurs is in the context of a binary arithmetic operation in which one of the operands is an image valued expression and the other operand is a scalar valued expression. In this case, the scalar operand is interpreted as a uniform implicit image having the same coordinate set as the actual image operand. The value of each pixel of this implicit image is equal to the value of the scalar operand. The arithmetic operation is carried out between the actual image operand and the implicit image operand in accordance with the definition of the particular operation in the Image Algebra.

The second way in which a scalar value denotes an implicit image is in the context of an arithmetic assignment statement. Specifically, if the expression on the right-hand side of the '=' symbol produces a scalar result and the left-hand side is an image, then the scalar value produced by the expression is interpreted as a uniform implicit image (i) with the same coordinate set as the actual image, and (ii) with the value of each pixel equal to the scalar

value. The effect of an arithmetic assignment statement of this form is for the implicit image to be assigned to the actual image in accordance with the rules of the Image Algebra. In particular, every pixel in the actual image is assigned the value of the corresponding pixel in the implicit image.

c. Arithmetic Function References

A reference to an arithmetic `FUNCTION` within an arithmetic expression of a FORTRAN 77 program always returns a scalar result. In IAF, a reference to an arithmetic `FUNCTION` within an arithmetic expression can also return an image result. This is effected by indicating, using an appropriate syntax, that the function is to be applied to one or more images in a point-wise fashion. The second example program of Section IV shows how this program construct is used.

A reference to an arithmetic function denotes an image (i.e., returns an image result) if all of the conditions listed below are satisfied.

- The function reference contains at least one actual argument.
- At least one of the actual arguments in the reference to the function is an image or an arithmetic expression that produces an image result. Such an actual argument is called an image actual argument.
- At least one of the image actual arguments in the reference to the function is prepended with the '`@`' symbol. This denotes the intention that the function is to be applied in a point-wise fashion to the image.
- For every image actual argument that is prepended with the '`@`' symbol, the corresponding dummy argument in the definition of the function is a scalar variable.

Only image actual arguments can be prepended with the '`@`' symbol. In particular, it is invalid to prepend an actual argument that is a scalar or an arithmetic expression that produces a scalar result with the '`@`' symbol.

An arithmetic function reference that denotes an image based on the above criteria is referred to as a *point-wise function reference*. A point-wise function reference can identify an `INTRINSIC` arithmetic function or a user-defined or library-supplied `EXTERNAL` arithmetic function. Note that a point-wise function reference can identify a function either by the name that appears in a `FUNCTION` statement or by the name that appears in an `ENTRY`

statement within a **FUNCTION** subprogram. The number, order and type of the actual arguments in a point-wise function reference must be in agreement with the number, order and type of the dummy arguments in the respective **FUNCTION** statement or **ENTRY** statement.

A point-wise function reference possesses a manifest coordinate set. The manifest coordinate set of a point-wise function reference is used to establish the compatibility of the image actual arguments and to determine the coordinate set of the result image. The manifest coordinate set of a point-wise function reference is the coordinate set of the image actual argument that occurs leftmost in the list of actual arguments of the point-wise function reference. It is required that the coordinate-set of every image actual argument of a point-wise function reference be the same as the manifest coordinate set.

The image that results from a point-wise function reference satisfies the conditions listed below.

- The coordinate set of the result image is the same as the manifest coordinate set of the point-wise function reference.
- The type of the result image is determined by the type of the scalar result that is returned by the function.
- The value of each pixel in the result image is determined by the value that is returned by the function when it is applied to the corresponding pixels of the image actual arguments that are prepended with the '@' symbol together with all of the other actual arguments.

If the name used to reference the function is the same as the generic name of a FORTRAN 77 **INTRINSIC** function, then the type of the result image is determined in the same way FORTRAN 77 determines the type returned by a generic **INTRINSIC** function. Specifically, the type(s) of the actual argument(s) in the point-wise function reference determine the type of the result returned by the function.

A point-wise function reference, in addition to one or more image actual arguments, may have zero or more actual arguments that are not prepended with the '@' symbol. Any such actual argument that is a syntactically and semantically valid FORTRAN 77 actual argument is also a valid IAF actual argument in a point-wise function reference. Note that this allows arbitrary expressions (including arithmetic, character, relational, and logical expressions), arrays, subprograms, etc. to be used freely as actual arguments in point-wise

function references. Furthermore, an arbitrary arithmetic expression that produces an image result may be used as an actual argument in a point-wise function reference without prepending it with the '@' symbol. This is equivalent to passing an array to the function which corresponds to the image that is produced by the expression. Actual arguments that are not prefixed with the '@' symbol are associated with their corresponding dummy arguments in the definition of the referenced function in the same way in which actual arguments and dummy arguments are associated in FORTRAN 77.

d. Arithmetic Array Element References

An arithmetic array element reference within an arithmetic expression in a FORTRAN 77 program always denotes a scalar value. In IAF, an arithmetic array element reference within an arithmetic expression can also denote an image. This functionality is analogous to that provided by point-wise function references. The second example of Section IV shows how this facility can be used to implement functions via table lookup, yielding faster execution speed of Image Algebra FORTRAN programs.

An arithmetic array element reference denotes an image if all of the conditions below are satisfied.

- At least one index expression of the arithmetic array element reference is an image or an arithmetic expression that produces an image result. Such an index expression is called an image index expression.
- Every image index expression of the arithmetic array element reference is prepended with the '@' symbol.
- Every image index expression is of `INTEGER` type.
- For every image index expression, the value of every pixel of the image produced by the image index expression lies within the bounds of the dimension of the array being indexed by that expression.

Only an image index expression can be prepended legitimately with the '@' symbol. In particular, it is invalid to prepend a scalar index expression with the '@' symbol. With respect to the last constraint listed above, the preprocessor does not generate code to perform execution-time bounds-checking. Furthermore, it is not possible for the preprocessor to perform bounds-checking at translation time.

An arithmetic array element reference with one or more image index expressions is called an *image-value-indexed array reference*. The number of index expressions in an image-value-indexed array reference must be equal to the number of dimensions of the underlying array. All of the index expressions in an image-value-indexed array reference must be integer expressions.

An image-value-indexed array reference possesses a manifest coordinate set. The manifest coordinate set of an image-value-indexed array reference is used to establish the compatibility of the image indexing expressions and to determine the coordinate set of the result image. The manifest coordinate set of an image-value-indexed array reference is the coordinate set of the image index expression that occurs leftmost in the list of index expressions of the array reference. It is required that the coordinate set of every image index expression of an image-value-indexed array reference be the same as this manifest coordinate set.

The image denoted by an image-value-indexed array reference satisfies the conditions listed below.

- The coordinate set of the image is the same as the manifest coordinate set of the image-value-indexed array reference.
- The type of the result image is the same as the type of the underlying referenced array.
- The value of each pixel in the result image is determined by the value of the array element that is indexed by the values of the corresponding pixels of the image indexing expressions (each necessarily prepended with the '@' symbol) together with all of the other scalar index expressions.

The type of the array named in an image-value-indexed array reference must be a type that is valid for an image. The array named in an image-value-indexed array reference may have from one to seven dimensions. Furthermore, the array named in an image-value-indexed array reference may be an adjustable array or an assumed-size array. An image-value-indexed array reference cannot appear on the left-hand side of the '=' symbol in an arithmetic assignment statement.

In essence, an image-value-indexed array reference models a partial function that is effectively implemented by table look-up. More specifically, the domain of the function is

simply the dimension bounds of the underlying array. The domain over which the function is defined (its effective domain) depends on the values of the index expressions in the image-value-indexed array reference. The range of the function is determined by the values that are assigned to the various elements of the underlying array. Therefore, the effective domain and range are dynamic since they are established at execution-time. In that respect, an array that is used in an image-value-indexed array reference models a family of partial functions.

It is emphasized that the '@' symbol can be applied legitimately only to an image actual argument that occurs in a reference to an arithmetic **FUNCTION** or to an image index expression that occurs in an arithmetic array element reference. In that respect, the '@' symbol is not a new arithmetic operator that can be arbitrarily embedded within an arithmetic expression as a unary operator. It is more correctly viewed as a directive to the preprocessor that instructs it to interpret a particular arithmetic function reference or arithmetic array element reference in a special way.

3. IMAGE ALGEBRA FORTRAN ARITHMETIC OPERATORS

The Image Algebra extensions of IAF only affect arithmetic expressions. In that respect, the semantics of the FORTRAN 77 arithmetic operators are extended in IAF to support the use of images as operands. In addition, several new arithmetic operators are available in IAF that explicitly require at least one image operand. Finally, image-template operations are incorporated into arithmetic expressions as particularly powerful mechanisms for effecting a virtually limitless assortment of image transformations.

We first discuss the further overloading of FORTRAN 77 arithmetic operators in IAF. Next we describe the new arithmetic operators that are peculiar to IAF. Finally, template declarations, template definitions, and the IAF image-template operators are presented. A Backus-Naur Form (BNF) grammar which describes the syntax of IAF arithmetic expressions is given in Appendix A.

In the following, a *scalar* operand is one of the following: (i) an unsigned arithmetic constant, (ii) a symbolic name of an arithmetic constant (i.e., defined in a **PARAMETER** statement), (iii) an arithmetic variable reference, (iv) an arithmetic array element reference, (v) an arithmetic function reference, (vi) an arithmetic expression that produces a scalar result, or (vii) a parenthesized arithmetic expression that produces a scalar result. An *image* operand is one of the following: (i) an unsubscripted arithmetic array reference, (ii) an arithmetic

image-value-indexed array reference, (iii) an arithmetic point-wise function reference, (iv) an arithmetic expression that produces an image result, or (v) a parenthesized arithmetic expression that produces an image result. An arbitrary expression produces a scalar result or an image result depending on the specific operands and operations involved and the relative precedence of the operators. Note that the type of each operand of an arithmetic operation must be one of the built-in FORTRAN 77 arithmetic types (i.e., `INTEGER`, `REAL`, `DOUBLE PRECISION`, or `COMPLEX`).

a. FORTRAN 77 Arithmetic Operators

The arithmetic operators that are native to FORTRAN 77 are exponentiation '**', division '/', multiplication '*', binary subtraction and unary negation '-', and binary addition and unary identity '+'. In FORTRAN 77, all of these operators require scalar operands and produce scalar results. In IAF, the native FORTRAN 77 unary operators can operate on either a scalar or an image operand, whereas the native FORTRAN 77 binary operators can operate on any combination of two scalar and image operands. Accordingly, in IAF these arithmetic operators can produce either scalar or image results depending on the operands that they are applied to.

(1) Unary Operators

The unary arithmetic operators that are native to FORTRAN 77 are negation '-' and identity '+'. The unary '-' and '+' operators can be applied to either scalar operands or image operands.

The semantics of the IAF negation operator '-' applied to a scalar operand are exactly the same as the semantics of the FORTRAN 77 negation operator.

When the negation operator is applied to an image operand, the result of the operation is an image. The manifest coordinate set of a unary negation operation that has an image operand is the coordinate set of the image operand. The coordinate set of the result image is the same as the manifest coordinate set of the operation. The type of the result image is the same as the type of the image operand. The effect of the negation operator is to negate the value of each pixel of the image operand.

The IAF identity operator '+' is treated as a null operation, regardless of whether the single operand involved is a scalar operand or an image operand. More specifically, the unary '+' has no effect on its single operand.

(2) Binary Operators

The binary arithmetic operators that are native to FORTRAN 77 are exponentiation '**', division '/', multiplication '*', subtraction '-', and addition '+'. In IAF, these binary operators can be applied to any combination of two arithmetic scalar and image operands. In the following, "*bop*" is used to denote any one of the five binary arithmetic operators '**', '/', '*', '-', or '+'.

If the two operands of a "*bop*" operator are both scalar operands, then the semantics of the corresponding operation are exactly the same as the semantics of the operation in FORTRAN 77.

If a "*bop*" operator has one image operand and one scalar operand, then the scalar operand is treated as an implicit image. The result of a "*bop*" operation with one image operand and one scalar operand is an image. The manifest coordinate set of the "*bop*" operation is the coordinate set of the image operand. The coordinate set of the implicit image is the same as the manifest coordinate set of the operation. The coordinate set of the result image is the same as the manifest coordinate set of the operation.

If both operands of a "*bop*" operation are images, then the result of the operation is an image. The manifest coordinate set of the "*bop*" operation is the coordinate set of the left image operand. The coordinate set of the right image operand must be the same as the manifest coordinate set of the operation. The coordinate set of the result image is the same as the manifest coordinate set of the operation.

Regardless of whether a "*bop*" operator has one or two image operands, the operation is performed as if both operands are images since a scalar operand that accompanies an image operand is promoted to an image (albeit, an implicit image) with the same coordinate set as the image operand. The result image is constructed by applying the "*bop*" operator to the two image operands in a pixel-wise fashion. More specifically, the value of each pixel in the result image is determined from the result of the "*bop*" operation applied to the corresponding pixels of the images that constitute the left and right operands.

Regardless of whether a "*bop*" operator has one or two image operands, the type of the image result that is produced by the operation is determined by the same rules that are used in FORTRAN 77 for determining the result type of a "*bop*" operation that has two scalar operands. Thus, the type of the result image is determined as if both operands were scalar operands.

The definition of the exponentiation '**' operator in IAF departs slightly from its definition in FORTRAN 77. In FORTRAN 77 0^k is undefined for all $k \leq 0$, however, we give such expressions a meaning in IAF. In IAF, the following rules define the exponentiation operator: for arbitrary scalar operand a, $a^0 = 1$ if and only if $a \neq 0$; if $a = 0$, then the operation a^0 is undefined; and for arbitrary scalar operand k, $0^k = 0$ for all $k \neq 0$. These rules for exponentiation extend readily to image operands.

b. Additional IAF Arithmetic Operators

In addition to the arithmetic operators that are native to FORTRAN 77, several other arithmetic operators are provided in IAF. In particular, four new unary arithmetic operators, three new binary arithmetic operators, and six characteristic function operators are available in IAF. Each of these operators require at least one image operand. The unary operators are discussed first, followed by the binary operators and the characteristic function operators.

(1) Unary Image Operators

The four unary arithmetic operators that are unique to IAF are image pseudo-inverse '`!inv`', image maximum '`!max`', image minimum '`!min`', and image sum '`!sum`'. The single arithmetic operand of each of these operators must be an image.

The unary '`!inv`' operator implements the Image Algebra pseudo-inverse operation. When the '`!inv`' operator is applied to an image operand, the result of the operation is another image. The type of the result image is the same as the type of the image operand. The coordinate set of the result image is the same as the coordinate set of the image operand. The effect of the '`!inv`' operator is to compute the pseudo-inverse of the image operand. The pseudo inverse of an image **a** is defined as `!inva` ≡ `a**-1`.

The unary '`!max`' operator implements the Image Algebra unary image-maximum operation ($\bigvee$). When the '`!max`' operator is applied to an image operand, the result of the operation is a scalar value. The type of the scalar result is the same as the type of the image operand. The '`!max`' operator produces the value of the pixel in the image operand that has the greatest value. The unary '`!max`' operator cannot be applied to a `COMPLEX` image.

The unary '`!min`' operator implements the Image Algebra unary image-minimum operation ($\bigwedge$). When the '`!min`' operator is applied to an image operand, the

result of the operation is a scalar value. The type of the scalar result is the same as the type of the image operand. The '**!min**' operator produces the value of the pixel in the image operand that has the least value. The unary '**!min**' operator cannot be applied to a COMPLEX image.

The unary '**!sum**' operator implements the Image Algebra unary image-sum operation ($\sum$). When the '**!sum**' operator is applied to an image operand, the result of the operation is a scalar value. The type of the scalar result is the same as the type of the image operand. The '**!sum**' operator produces the scalar quantity that is obtained from adding the values of all of the pixels in the image operand.

(2) Binary Image-Image Operators

The three binary arithmetic operators that are unique to IAF are image dot-product '**!dot**', binary image-maximum '**!max**', and binary image-minimum '**!min**'. At least one of the operands of each of these operators must be an image.

If exactly one of the operands of a '**!dot**', binary '**!max**', or binary '**!min**' operator is an image and the other is a scalar, then the scalar operand is treated as an implicit image. The manifest coordinate set of the operation is the coordinate set of the image operand. The coordinate set of the implicit image is the same as the manifest coordinate set of the operation.

If both operands of a '**!dot**', binary '**!max**', or binary '**!min**' operator are images, then the manifest coordinate set of the operation is the coordinate set of the left image operand. The coordinate set of the right image operand must be the same as the manifest coordinate set of the operation.

Regardless of whether a '**!dot**', binary '**!max**', or binary '**!min**' operator has one or two image operands, the operation is performed as if both operands are images since a scalar operand that accompanies an image operand is promoted to an image with the same coordinate set as the image operand.

The type of the result of a '**!dot**', binary '**!max**', or binary '**!min**' operation is determined as if the two operands were simple scalar variables and the operator was replaced by any one of the FORTRAN 77 binary arithmetic operators '/', '*', '-', or '+'.

The result of the '**!dot**' operation is a scalar value. The scalar quantity is produced by effecting the Image Algebra dot-product operation on the two image operands.

The result of a binary '**!max**' or binary '**!min**' operation is an image. The coordinate set of the image result is the same as the manifest coordinate set of the operation. For the binary '**!max**' (resp. '**!min**') operation, the value of each pixel in the result image is set to the maximum (resp. minimum) of the two corresponding pixels in the operand images.

(3) Characteristic-Function Operators

The six binary characteristic-function operators available in IAF are '**<**' (less than), '**<=**' (less than or equal to), '**==**' (equal to), '**!=**' (not equal to), '**>**' (greater than), and '**>=**' (greater than or equal to). The '**<**' (resp. '**<=**', '**==**', '**!=**', '**>**', '**>=**') operator implements the Image Algebra $\chi_{<}$ (resp. $\chi_{\leq}$, $\chi_{=}$, $\chi_{\neq}$, $\chi_{>}$, $\chi_{\geq}$) operation. At least one of the operands of each of these operators must be an image.

If exactly one of the operands of a characteristic-function operator is an image and the other is a scalar, then the scalar operand is treated as an implicit image. The manifest coordinate set of the operation is the coordinate set of the image operand. The coordinate set of the implicit image is the same as the manifest coordinate set of the operation.

If both operands of a characteristic-function operator are images, then the manifest coordinate set of the operation is the coordinate set of the left image operand. The coordinate set of the right image operand must be the same as the manifest coordinate set of the operation.

Regardless of whether a characteristic-function operator has one or two image operands, the operation is performed as if both operands are images since a scalar operand that accompanies an image operand is effectively "promoted" to an image with the same coordinate set as the image operand.

The result of every characteristic-function operation is an image of **INTEGER** type. The coordinate set of the image result is the same as the manifest coordinate set of the operation. With respect to the '**<**' (resp. '**<=**', '**==**', '**!=**', '**>**', '**>=**') characteristic-function operation, each pixel in the result image for which the values of the corresponding pixels in the two image operands are related by the '**<**' (resp. '**<=**', '**==**', '**!=**', '**>**', '**>=**') relation is set to one. All of the other pixels in the result image are set to zero.

Neither of the two operands of a characteristic-function operator can be of **COMPLEX** type.

4. TEMPLATES

In terms of image processing applications, templates and image-template operations are the most powerful tools of IAF. The functionality provided by Image Algebra templates is implemented in IAF by a combination of template *declarations*, template *definitions*, and image-template *operations*. A template is *declared* in a template declaration statement. A template declaration statement is a nonexecutable statement that is unique to IAF and has no counterpart in FORTRAN 77. A template is *defined* in a template definition. A template definition is a new kind of program unit that is syntactically similar to a **FUNCTION** definition. A template is *referenced* in an image-template operation that can occur in an arbitrary arithmetic expression. Template declarations are discussed first, followed by template definitions and image-template operations.

a. Template Declarations

Every template referenced in an image-template operation in a program unit must be declared within that same program unit. There are two basic kinds of templates, translation-*invariant* templates and translation-*variant* templates. Consequently, the template declaration statement has two basic forms. The syntax of IAF template declaration statements are illustrated as follows.

[type-specifier] **INVARIANT** IDENT [, IDENT ...]
[type-specifier] **VARIANT** IDENT [, IDENT ...]

A template declaration statement consists of an optional type specifier, followed by one of the keywords **INVARIANT** or **VARIANT**, followed by a list of one or more identifiers separated by commas. Each identifier appearing in a template declaration statement is the name of a template that may be referenced subsequently in an image-template operation within the same program unit. The following code fragment illustrates the declaration of four templates.

```
INTEGER INVARIANT sobel
VARIANT avg1, avg2, avg3
```

A template can be declared in a template declaration statement at most once in a program unit. An identifier that is declared as the name of a template in a template declaration statement cannot appear in any other declaration statement within the same program unit.

The valid template types are **INTEGER**, **REAL**, **DOUBLE PRECISION**, and **COMPLEX**. In the absence of an explicit type specification for a template, the usual FORTRAN 77 rules for implicit typing apply. In particular, a type specifier is used in a template declaration statement to confirm or override the implicit type for the template names that appear in the statement. The type of a template cannot be specified in a separate type-statement. The type given to a template (either explicitly or implicitly) in a template declaration statement must be the same as the type given to the template (either explicitly or implicitly) in its definition.

A template declaration statement is nonexecutable. All template declarations in a program unit must occur before the first executable statement of the program unit. The declaration of a template within a program unit in which the template is not referenced in any image-template operation within an arithmetic expression has no affect.

b. Template Definitions

Every template that is referenced in an image-template operation within an executable program must have a corresponding template definition. A new kind of program unit is available in IAF for the express purpose of defining templates. The program units that are used to define templates are known as template definitions. As is the case for the types of program units that are standard to FORTRAN 77 (e.g., **FUNCTION** subprograms, **SUBROUTINE** subprograms, etc.), template definitions can be compiled separately from the other program units that comprise an executable program in which they are used Thus, archived libraries of templates can be conveniently created.

An IAF template definition is syntactically quite similar to the definition of a **FUNCTION** in FORTRAN 77. The first statement of a template definition must be a **TEMPLATE** statement and the last statement of a template definition must be an **END** statement. Between the leading **TEMPLATE** statement and the trailing **END** statement is a sequence of FORTRAN 77 statements and template-weight assignment statements.

The **TEMPLATE** statement of a template definition identifies the name of the template and the template dummy arguments, if any. In addition, the type of the template may be optionally specified. The identifier that names a template in a **TEMPLATE** statement is the same name that is used to reference the template in image-template operations that occur in arithmetic expressions in other program units of the executable program. The syntax of an IAF **TEMPLATE** statement is shown here.

[type-specifier] **TEMPLATE** IDENT ([dummy arguments])

A **TEMPLATE** statement consists of an optional type specifier, followed by the keyword **TEMPLATE**, followed by the identifier that names the template, followed by a parenthesized list of template dummy arguments.

The name of a template that is given in the **TEMPLATE** statement cannot be the same as the name of a **FUNCTION** or **SUBROUTINE**, or the same as the name of an **ENTRY** within the definition of a **FUNCTION** or **SUBROUTINE** within the same executable program. However, the name of a template can be used as the name of a variable, array, statement function, or symbolic name of a constant within another program unit of the executable program in which it is not referenced as a template in an image-template operation.

The valid template types are **INTEGER**, **REAL**, **DOUBLE PRECISION**, and **COMPLEX**. In the absence of an explicit type specification for a template, the usual FORTRAN 77 rules for implicit typing apply. In particular, a type specifier is used in a template definition statement to confirm or override the implicit type of the template. The type of a template cannot be specified in a separate type-statement within the body of the template definition.

A **TEMPLATE** statement may specify zero or more dummy arguments. The rules for the specification of dummy arguments and their subsequent optional declaration are the same as those which apply to the specification and optional declaration of dummy arguments within a FORTRAN 77 **FUNCTION** definition.

The statements that occur between the leading **TEMPLATE** statement and the trailing **END** statement of a template definition constitute the body of the template definition. The body of a template definition consists of an arbitrary number of FORTRAN 77 statements and one or more template-weight assignment statements. A template-weight assignment statement is a particular kind of assignment statement that is peculiar to IAF template definitions. Template-weight assignment statements are described shortly.

Aside from the noted exceptions, any valid FORTRAN 77 statement may occur within the body of a template definition. The following FORTRAN 77 statements cannot occur within the body of a template definition.

- A **PROGRAM** statement,

- a **BLOCK DATA** statement,
- a **SUBROUTINE** statement,
- a **FUNCTION** statement,
- **ENTRY** statement, or
- **END** statement.

None of the FORTRAN 77 statements occurring in the body of a template definition can make use of the Image Algebra extensions to IAF arithmetic expressions. A template definition cannot contain a template declaration statement or a **TEMPLATE** statement.

A template-weight assignment statement is a special kind of assignment statement that is peculiar to the definitions of templates in IAF. A template-weight assignment statement is an executable statement. A template-weight assignment statement consists of a subscripted reference to the template on the left-hand side of an '=' symbol and an arbitrary FORTRAN 77 arithmetic expression on the right-hand side. The left-hand side of a template-weight assignment statement is syntactically identical to an array element reference with two indexing expressions. The left-hand side of a template-weight assignment statement corresponds to a particular point in the configuration of the template. The expression on the right-hand side denotes the weight that is to be assigned to that point in the configuration of the template.

At the time of an image-template operation, the left-hand side of a template-weight assignment statement within the relevant template definition denotes a particular point in the source image of the image-template operation. Similarly, the expression on the right-hand side is the weight that becomes associated with that point. This point in the source image point is typically specified by suitable offsets with respect to the target pixel in the target image of the image-template operation. The **INTEGER** identifiers '**Y1**' and '**Y2**' are pseudo-variables that have special meaning within template definitions. Specifically, '**Y1**' and '**Y2**' index the target pixel in the target image of an image-template operation. The variable '**Y1**' corresponds to the row coordinate of the target pixel and '**Y2**' the column coordinate. The indexing expressions in the left-hand side of a template-weight assignment statement may be cast in terms of these pseudo-variables to establish an association between a point in the source image and the target pixel in the target image

It is prohibited for the weight associated with a particular point in the source image to be specified in more than one template-weight assignment statement in the body of a template definition. The values of 'Y1' and 'Y2' cannot be modified within a template definition either within an arithmetic assignment statement or as the side effect of a SUBROUTINE or FUNCTION reference. The arithmetic expression on the right-hand side of a template-weight assignment statement cannot make use of any of the Image Algebra extensions of IAF arithmetic expressions. The type of the result of the expression on the right-hand side must be assignment-compatible with the type of the template. The pseudo-identifiers 'Y1' and 'Y2' must not be declared in a type statement in a template definition. Suitable declarations for them are supplied by the preprocessor. If redundant declarations for them are mistakenly supplied by the user, such definitions will conflict with those that are automatically supplied by the preprocessor.

c. Image-Template Operators

There are five image-template operators available in IAF. These operators are generalized convolution '!gcon', additive maximum '!amax', additive minimum '!amin', multiplicative maximum '!mmax', and multiplicative minimum '!mmin'. The '!gcon' (resp. '!amax', '!amin', '!mmax', '!mmin') operator implements the Image Algebra $\oplus$ (resp. $\boxtimes$, $\boxtimes$, $\bigtriangledown$, $\bigtriangleup$) operation. The left operand of each of these operators must be an image. The right operand of each of these operators must be a valid reference to a template. The template that is referenced in an image-template operation must be declared in a template declaration statement in the program unit in which the operation occurs.

An image-template operation has a manifest coordinate set. If the image-template operation appears in the context of an arithmetic expression that appears on the right-hand side of the '=' symbol of an arithmetic assignment statement and the left-hand side is an image, then the manifest coordinate set of the image-template operation is the coordinate set of the image on the left-hand side. If an image-template operation appears in any other context, then the manifest coordinate set of the operation is the coordinate set of the image that is the left operand of the image-template operation.

A reference to a template in an image-template operation is syntactically similar to a reference to a FUNCTION or variable. If the template that is referenced in an image-template operation is defined with no template dummy arguments, then the template reference may either consist of (i) the name of the template, or (ii) the name of the template

followed by an empty pair of matching parentheses. If the template that is referenced in an image-template operation is defined with one or more template dummy arguments, then the template reference must consist of the name of the template followed by a parenthesized list of actual template arguments. The actual template arguments that are supplied in the template reference must agree in order, number, and type with the template dummy arguments that are specified in the template definition. Any argument that constitutes a valid FORTRAN 77 `FUNCTION` actual argument is also a valid template actual argument. Note that this includes using the names of subprograms and arrays as template actual arguments. However, an arithmetic expression that occurs in a template actual argument cannot contain any of the Image Algebra extensions of IAF arithmetic expressions. Furthermore, a template cannot be passed to a template as an actual argument in a template reference.

The result of an image-template operation is an image. The coordinate set of the result image is the manifest coordinate set of the operation. The type of the result of an image-template operation is determined as if the two operands were simple scalar variables and the image-template operator was replaced by any one of the FORTRAN 77 binary arithmetic operators '/', '*', '-', or '+'.

5. IMAGE ALGEBRA EXPRESSIONS AND THEIR USE

Table 5 gives the precedence hierarchy of the entire set of IAF operators. Unless indicated to the contrary, an operator represents a binary operation. Operators that are explicitly superscripted with a 1 (resp. 2) denote unary (resp. binary) operations.

Table 5. IAF OPERATOR PRECEDENCE HIERARCHY

Precedence of IAF Operators
** (*Highest*)
!gcon !amax !amin !mmax !mmin
* /
!dot !max[2] !min[2]
+[1,2] -[1,2] !inv[1] !max[1] !min[1] !sum[1] //
< <= == != > >=
.LT. .LE. .EQ. .NE. .GT. .GE.
.NOT.[1]
.AND.
.OR.
.EQV. .NEQV. (*Lowest*)

The Image Algebra extensions incorporated into IAF primarily affect arithmetic expressions. Consequently, the semantics and rules of use of FORTRAN 77 character, relational, and logical operators and expressions are exactly the same as their semantics and rules of use in IAF with the following exception. Arbitrary arithmetic expressions that make full use of the Image Algebra extensions may be used as operands of relational operators within relational expressions as long as all such arithmetic expressions produce scalar results. In particular, the FORTRAN 77 relational operators '.LT.', '.LE.', '.EQ.', '.NE.', '.GT.', and '.GE.' are not defined for image operands. Relational expressions are used, in turn, as operands of logical operators in logical expressions. This extended form of relational expressions may be freely used within arbitrary logical expressions, as well.

a. Valid Usage

Arithmetic expressions that appear in the following FORTRAN 77 executable statements may make full use of the Image Algebra extensions of IAF arithmetic expressions under the constraints outlined below unless the statement is the terminal statement of a DO-loop.

- arithmetic assignment statements,
- logical assignment statements,
- block IF statements,
- ELSE IF statements,

- CALL statements,
- DO statements, and
- arithmetic IF statements.

A statement that is the terminal statement of a DO-loop cannot contain an arithmetic expression that makes use of the Image Algebra extensions imparted to arithmetic expressions.

All IAF statements that are one of the above types are parsed by the IAF preprocessor. An IAF statement that is one of the above types and which is found to contain Image Algebra extensions is translated by the preprocessor into one or more semantically equivalent FORTRAN 77 statements. An IAF statement that is one of the above types and which is found not to contain Image Algebra extensions is simply echoed to the standard output file. Statements that do not require any translation are still fully parsed by the preprocessor. Thus, FORTRAN 77 lexical and syntactic errors are detected and appropriate diagnostic error messages are emitted.

In IAF, the left-hand side of an arithmetic assignment statement can either be (i) a variable name, (ii) an arithmetic array element reference, or (iii) an unsubscripted array name. If the left-hand side of an arithmetic assignment statement is either (i) a variable name or (ii) an arithmetic array element reference, then the expression on the right-hand side must produce a scalar result. More specifically, no restriction is placed on the use of the Image Algebra extensions in the expression on the right-hand side provided that the expression produces a scalar result. Note that intermediate results of the right-hand side expression may produce image results. If the left-hand side of an arithmetic assignment statement is an array name, then the array represents an image. An assignment statement that has an image on the left-hand side has a manifest coordinate set that is the coordinate set of the left-hand side image. In this case, the expression on the right-hand side can produce either a scalar result or an image result. If the expression on the right-hand side produces a scalar result, then it is treated as an implicit image with the same coordinate set as the manifest coordinate set of the assignment operation. If the expression on the right-hand side produces an image result, then the coordinate set of the right-hand side image result must be the same as the manifest coordinate set of the assignment operation.

The logical expression that occurs on the right-hand side of the '=' symbol in a logical assignment statement may contain one or more arithmetic expressions (since logical

expressions can be composed of relational expressions which can, in turn, be composed of arithmetic expressions). In addition, the logical conditional expression that occurs in a block IF statement or an ELSE IF statement may also contain one or more arithmetic expressions. No restriction is placed on the use of the Image Algebra extensions in arithmetic expressions that occur in these contexts other than that each such arithmetic expression must produce a scalar result.

No restriction is placed on the use of the Image Algebra extensions in arithmetic expressions that occur in actual arguments that are passed to SUBROUTINEs in CALL statements. Note that this allows for the use of actual arguments that are arithmetic expressions which produce image results. Actual arguments that are arithmetic expressions which produce image results are treated as if they were arrays and are passed to the referenced SUBROUTINE like an array is passed. Note that the '@' symbol has no relevance whatsoever to actual arguments that appear in CALL statements. The '@' symbol is only relevant to actual arguments appearing in a reference to a FUNCTION.

No restriction is placed on the use of the Image Algebra extensions in the controlling expressions of a DO statement except that (i) each expression must produce a scalar result, and (ii) the type of the result of each expression must be either INTEGER, REAL, or DOUBLE PRECISION. The use of any of the Image Algebra extensions of arithmetic expressions and arithmetic assignment statements within the terminal statement of a DO-loop is prohibited.

No restriction is placed on the use of the Image Algebra extensions in the arithmetic expression of an arithmetic IF statement except that (i) the expression must produce a scalar result, and (ii) the type of the result produced by the expression must be either INTEGER, REAL, or DOUBLE PRECISION.

b. Invalid Usage

Arithmetic expressions may also appear in the following FORTRAN 77 executable statements:

- logical IF statements,
- computed GO TO statements,
- WRITE statements,

- **PRINT** statements,
- **RETURN** statements, and
- statement function statements.

None of the Image Algebra extensions imparted to arithmetic expressions can be employed in any of these statements in IAF. None of these statements are parsed by the IAF preprocessor. They are simply echoed to the standard output, as is, and otherwise ignored. With respect to the logical **IF** statement, these restrictions apply to the logical conditional expression of the statement, as well as to any arithmetic expression occurring in the conditionally-executed statement of the logical **IF** statement.

c. Temporary Storage Management

IAF arithmetic expressions that make use of the Image Algebra extensions of IAF are translated into equivalent FORTRAN 77 expressions. These transformations will, in general, involve the employment of temporary variables for the purpose of storing intermediate results of expression evaluations. The intermediate values that arise during the evaluation of an expression will typically be some combination of scalars and images. The IAF preprocessor manages the allocation and use of any temporary storage that is required to effect the translation of an Image Algebra-enhanced arithmetic expression into equivalent FORTRAN 77 arithmetic expressions.

The IAF programmer may choose to utilize one or more temporary scalar or image entities for the purpose of retaining some scalar or image value of interest, improving the efficiency of an algorithm, or to circumvent a restriction imposed on the use of the Image Algebra extensions of arithmetic expressions. However, the IAF programmer is in no way explicitly burdened with the responsibility of managing temporary storage space. In that respect, IAF arithmetic expressions can be written in as natural a manner as is allowed by their syntax and use within IAF statements.

6. FURTHER RESTRICTIONS AND EXCEPTIONS

Other restrictions and exceptions on the use of IAF that have not been mentioned previously are enumerated here.

a. Reserved Name Space

Each valid IAF statement occurring in an IAF program is translated by the IAF preprocessor into a sequence of semantically-equivalent valid FORTRAN 77 statements. In the course of translation, the preprocessor defines and uses various identifiers (potentially many). Hence we reserve a portion of the FORTRAN 77 identifier name space for the preprocessor to accommodate its needs. Specifically, all identifiers that are six characters in length and which start with 'XX' are reserved for this purpose. A diagnostic warning message is issued if the programmer uses an identifier that encroaches on the reserved name space. A dictionary of preprocessor-generated names, including a brief description of their meaning, is given in Appendix B.

b. Reserved Statement Labels

During translation, many of the statements that are generated by the preprocessor require a statement label. To avoid potential conflicts with statement labels used by the programmer, a portion of the statement-label space, specifically the sequence **99000** to **99999** inclusive, is reserved for preprocessor-generated statement labels. A diagnostic warning message is issued if the programmer uses a statement label that encroaches on the reserved space of statement labels.

c. Features Non-Standard to FORTRAN 77

Several non-ANSI Standard features that are included in many implementations of FORTRAN 77 are also recognized by the preprocessor. In the following, a description of the non-standard features specifically allowed by the preprocessor is given.

- The IAF preprocessor is case insensitive. However, during the translation of each statement, the preprocessor maps all lower-case letters that occur in identifiers and keywords to upper-case. Identifiers and keywords are produced in upper-case in all code that is generated by the preprocessor. For this reason, keywords and identifiers originally entered by the programmer in lower-case may be reproduced in upper-case by the preprocessor. FORTRAN 77 keywords and IAF keywords and code fragments are transcribed in upper-case in this document solely to impart emphasis and consistency.
- Identifiers are considered significant only in their first 32 characters. Identifiers that exceed thirty-two characters are truncated to thirty-two characters. A diagnostic warning message is issued whenever the preprocessor is forced to truncate an identifier.

- Non-standard statements denoted by the keywords **INCLUDE** and **NAMELIST** are considered valid IAF statements. These statements are given no interpretation by the preprocessor and are simply echoed to the standard output.

- The non-standard type specifiers **INTEGER*2**, **INTEGER*4**, **LOGICAL*2**, and **LOGICAL*4** can be used within type-statements and **FUNCTION** statements to override or confirm the implicit type of identifiers. With respect to the Image Algebra extensions of IAF, the type specifier **INTEGER*2** can be used to override or confirm the implicit type of an array that is used as an image provided, of course, that this feature is supported by the resident FORTRAN 77 compiler. The type specifier **INTEGER*2 cannot** be used to override or confirm the implicit type of a template in either a template declaration statement or a template definition statement. The type specifier **INTEGER*4** is treated as an alias for the standard FORTRAN 77 type specifier **INTEGER**. The two type specifiers **LOGICAL*2** and **LOGICAL*4** are irrelevant to the Image Algebra extensions of IAF; both of these type specifiers are treated as aliases for the standard FORTRAN 77 type specifier **LOGICAL**. For mixed-mode arithmetic, the rules for determining the type of the result of an arithmetic operation are readily extended to include the **INTEGER*2** type as follows. If the operand of a unary arithmetic operation is of type **INTEGER*2**, then the result is of type **INTEGER*2**. If one and only one operand of a binary arithmetic operation is of type **INTEGER*2**, then the type of the result is the type of the operand that is not of type **INTEGER*2**. If both operands of a binary arithmetic operation is of type **INTEGER*2**, then the result is of type **INTEGER*2**. Note that since the declaration and definition of a template with type **INTEGER*2** is proscribed, the type of the result of an image-template operation can never be **INTEGER*2**.

- The following non-standard variants of the **IMPLICIT** statement are allowed for the specification of implicit data typing and are considered equivalent.

```
IMPLICIT NONE ( ... )
IMPLICIT UNDEFINED ( ... )
```

  The "..." denotes the usual letter group(s) of the standard **IMPLICIT** statement. Undeclared identifiers whose implicit type is either **NONE** or **UNDEFINED** will elicit a diagnostic error message if the identifier occurs in an executable statement that is parsed by the preprocessor.

- The character string length specification in a **CHARACTER** type-statement may be written using the symbolic name of an integer constant in either of the following forms.

```
CHARACTER*(CDLEN) CARD
CHARACTER*CDLEN CARD
```

  The ANSI Standard prohibits the second form by requiring that the character string length specification be enclosed within parentheses if it is not an integer constant.

- Hollerith constants may be used in **CALL** statements as actual arguments to subroutine subprograms and as initializers within **DATA** statements.

- An '**&**' in column one of an IAF source line is interpreted as a continuation character. Use of this feature does not affect the proper interpretation of continuation characters that are specified in the standard manner with a character other than a blank or '**0**' in column six.

- An ASCII horizontal-tab character that occurs in any of the first six columns of an IAF source line effectively shifts the character immediately following the horizontal-tab character to column seven and shifts all of the other characters remaining in the line a corresponding amount to the right. Therefore, if a horizontal-tab is placed in one of the first six columns of an IAF source line and the line is neither a comment nor a line continued with a '**&**' placed in column one, the remaining text in the line to the right of the horizontal-tab is treated as the first line of a new statement.

- An IAF source line that has any one of the characters '**C**', '*', or '**c**' in column one is interpreted as a comment line.

- An IAF source line with a '**#**' in column one is assumed to be relevant to the C preprocessor. Each such line is echoed to the standard output and otherwise ignored by the IAF preprocessor.

- Either one of the characters '**'**' or '**"**' may be used to enclose quoted character string constants. However, if '**'**' (resp. '**"**') is used as the opening quote character of a character string constant, then '**'**' (resp. '**"**') must be used as the closing quote character. The interpretation of quote characters occurring within a character constant follows the usual conventions whereby the occurrence of '**''**' (resp. '**""**') within a character constant that is enclosed within '**'**' (resp. '**"**') quote characters is interpreted as one '**'**' (resp. '**"**').

- The programmer may make full use of the ASCII character set within quoted character string constants and comments.

SECTION IV
EXAMPLES OF THE USE OF IMAGE ALGEBRA FORTRAN

This section gives example Image Algebra FORTRAN programs that solve a variety of image processing problems. Assume the existence of a function **rdimg** that reads a 64 by 64 image from some input file, and a function **wrimg** that writes a 64 by 64 image to some output file.

The first example shows how to threshold an image such that the resulting image will have value one at coordinates where the source image pixels have values in some range k_1 through k_2 while other coordinates in the result image have value zero.

```
program thresh
integer NROWS, NCOLS
parameter (NROWS=64,NCOLS=64)
integer image(NROWS,NCOLS)
integer k1, k2

call rdimg(image)
read *, k1, k2

image = ( image >= k1 ) * ( image <= k2 )

call wrimg( image)
end
```

Figure 12 shows the result of thresholding an image with gray values in the range zero to 31 to only those values lying between 5 and 25.

Figure 12. Image **a** and image **a** thresholded to range 5 to 25

The second example shows how to use the @ construct to apply a function pointwise to an image as well as to pointwise induce the subscripting of an array on an image. The program computes the sin of an image two ways: first by applying the sin function directly to an image, then by preparing a lookup table for the sin function on the gray value range of the source image and computing the sin by table lookup.

```
      program sinprog
      integer NROWS, NCOLS
      parameter (NROWS=64,NCOLS=64)
      integer image(NROWS,NCOLS),image2(NROWS,NCOLS)
      integer image3(NROWS,NCOLS)
      real stable(0:255)

      call rdimg(image)

C Directly compute the sin of the image
C
      image2 = 255*sin( @ real( @ image))

C Prepare a lookup table for the sin function on the gray value range
C
      do 10 i = 0, 255
            stable(i) = sin(real(i))
   10 continue

C Compute the sin by looking up values in the lookup table
C
      image3 = 255*stable( @image)

      call wrimg(image2)
      call wrimg(image3)
      end
```

The example that follows, in addition to showing further use of the @ construct, demonstrates the declaration and use of translation invariant templates. The program computes the familiar sobel edge operation. The templates sobel1 and sobel2 of the program are shown pictorially in Figure 13.

Figure 13. Sobel Templates **sobel1** and **sobel2**

Note that in the program listing that follows, the zeros in these neighborhoods do not appear in the template definition. Recall that zero template weights do not appear in the configuration of a template, hence zero weights need not be assigned in a template definition.

```
      program sobel
      parameter (nrows=64, ncols=64)
      integer image(nrows,ncols)
      integer invariant sobel1, sobel2

      call rdimg( image)

      image = nint( @ sqrt( @ real( @ (image !gcon sobel1)**2 +
     +                                (image !gcon sobel2)**2)))

      call wrimg( image)
      end

      integer template sobel1()
      sobel1( Y1+1, Y2-1) = 1
      sobel1( Y1+1, Y2  ) = 2
      sobel1( Y1+1, Y2+1) = 1
      sobel1( Y1-1, Y2-1) = -1
      sobel1( Y1-1, Y2  ) = -2
      sobel1( Y1-1, Y2+1) = -1
      end

      integer template sobel2()
      sobel2( Y1-1, Y2+1) = 1
      sobel2( Y1  , Y2+1  ) = 2
      sobel2( Y1+1, Y2+1) = 1
      sobel2( Y1-1, Y2-1) = -1
      sobel2( Y1  , Y2-1) = -2
      sobel2( Y1+1, Y2-1) = -1
      end
```

Figure 14 shows the result of execution of this program on an image.

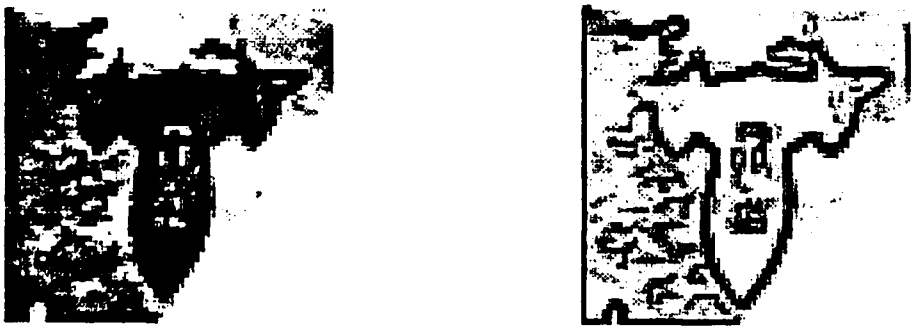

Figure 14. Image **a** and the Sobel edges of image **a**

The next example shows the definition and use of a translation invariant template that will translate an image in its coordinate set. Portions of the image that are translated outside the coordinate set in the resulting image are lost. Portions of the resulting image that have source pixels outside the source coordinate set take on the value zero.

```
      program translat
      integer NROWS, NCOLS
      parameter (NROWS=64,NCOLS=64)
      integer image(NROWS,NCOLS)
      integer invariant trans

      call rdimg( image)
      image = image !gcon trans(20,20)
      call wrimg( image)

      end

      INTEGER TEMPLATE TRANS( DELY1, DELY2)
C
C     Translates the image by DELY1 in the Y1 direction and DELY2 in the
C     Y2 direction, i.e. oldimage(0,0) ends up at newimage(DELY1,DELY2)
C
C     usage -- A !gcon TRANS( DELY1, DELY2)
C
C
      INTEGER DELY1, DELY2
      TRANS( Y1 - DELY2, Y2 - DELY2) = 1
      END
```

Note the use of pseudo-variables **Y1** and **Y2** representing the row and column pixel locations repectively of the target pixel in the definition of the template **trans**. This invariant template definition indicates that the source configuration for a point (y_1, y_2) consists of the single point $(y_1-\Delta y_1, y_2-\Delta y_2)$. This means that the pixel at coordinate $(\Delta y_1, \Delta y_2)$ in the result image will receive its value from the source image pixel at coordinate (0,0), for example. Figure 15 shows the result of execution of this program.

Figure 15. Image **a** and image **a !gcon trans**(20, 20)

The following example uses two translation variant templates, one of which is used to "matte" out a region of an image, the other to "isolate" that same region. The values in the isolated region are subtracted from the maximum image value (giving a negative-like effect). In this case, the configuration of the templates matte and isolate varies depending on the target pixel location and each configuration contains either a single pixel or is empty.

```
      program variants
      integer NROWS, NCOLS
      parameter (NROWS=64,NCOLS=64)
      integer image(NROWS,NCOLS)
      integer variant matte, isolate

      call rdimg(image)

      image = image !gcon matte( 4, 4, 32,32)
     +        +((!max image) - image) !gcon isolate( 4, 4, 32, 32)

      call wrimg( image)
      end
```

```
      INTEGER TEMPLATE ISOLATE( Y1LOW, Y2LOW, Y1HIGH, Y2HIGH)
C
C     ISOLATE isolates all those pixels in an image lying
C     within the rectangle with lower left corner at Y1LOW, Y2LOW
C     and upper right corner at Y1HIGH, Y2HIGH
C
C     Usage : A !gcon ISOLATE( Y1LOW, Y2LOW, Y1HIGH, Y2HIGH)
C
      INTEGER Y1LOW, Y2LOW, Y1HIGH, Y2HIGH

      IF (Y1.GE.Y1LOW.AND.Y1.LE.Y1HIGH.AND.Y2.GE.Y2LOW.AND.Y2.LE.Y2HIGH)
     +   THEN
         ISOLATE( Y1, Y2) = 1
      ENDIF
      END

      INTEGER TEMPLATE MATTE( Y1LOW, Y2LOW, Y1HIGH, Y2HIGH)
C
C     MATTE mattes out all those pixels in an image lying
C     within the rectangle with lower left corner at Y1LOW, Y2LOW
C     and upper right corner at Y1HIGH, Y2HIGH
C
C     Usage : A !gcon MATTE( Y1LOW, Y2LOW, Y1HIGH, Y2HIGH)
C
      INTEGER Y1LOW, Y2LOW, Y1HIGH, Y2HIGH

      IF (Y1.LT.Y1LOW.OR.Y1.GT.Y1HIGH.OR.Y2.LT.Y2LOW.OR.Y2.GT.Y2HIGH)
     +   THEN
         MATTE( Y1, Y2) = 1
      ENDIF
      END
```

The result of execution of program **variants** is shown in Figure 16.

Figure 16. Image **a** and result of program **variants** on **a**

The next program uses an interesting pair of templates, **y1temp** and **y2temp**. When template **y1temp** is evaluated at a point $\mathbf{y} = (y_1, y_2)$, it has a configuration containing the single pixel $\mathbf{y}$ and having weight y_1. Similarly, **y2temp**, when evaluated at a point $\mathbf{y} = (y_1, y_2)$ has a configuration containing the single pixel $\mathbf{y}$ and having weight y_2. The program **centroid** uses these templates to compute the centroid of a binary image, that is, an image with pixel values zero and one exclusively.

```
program centroid
integer NROWS, NCOLS
parameter (NROWS=64,NCOLS=64)
integer image(NROWS,NCOLS)
integer variant y1temp, y2temp
integer y1, y2

call rdimg(image)

y1 = (!sum (image !gcon y1temp))/(!sum image)
y2  = (!sum (image !gcon y2temp))/(!sum image)
print *, 'centroid is (', y1,',',y2,')'

image(y1,y2) = 0

call wrimg(image)
end

integer template y1temp()
y1temp(y1,y2) = y1
end

integer template y2temp()
y2temp(y1,y2) = y2
end
```

Note that this program uses variables named **Y1** and **Y2** outside the context of a template definition. In such a case, these variable names have no special significance. Such use of these variable names is permitted. Figure 17 shows the behavior of this centroid finding program on a binary image. In the result image, the centroid is marked by the black pixel. The rest of the non-zero pixels in the image are gray.

Figure 17. Binary image **a** and image showing computed centroid of **a**

The final example shows how one might define a template that can be used to rotate images through arbitrary angles. The method of rotation uses a simplistic model of pixel gray values to perform interpolation of gray values. It assumes that pixels are square patches with uniform gray value. In performing the rotation, every pixel **y** in the target coordinate set is given a value derived by summing the interpolation weights of the template configuration assigned to **y** by the rotate template. The configuration contains the four points closest to the inverse rotation of pixel **y**. The interpolation weights are approximate.

```
C
C       rotate.iaf
C
C  demonstrate an averaging rotation using the generalize
C  convolution operation
C
        integer NROWS, NCOLS
        parameter (NROWS=64,NCOLS=64)
        integer image(NROWS,NCOLS)
        real theta, pi
        integer invariant trans
        variant rotate
C  A particularly useful number
        pi = atan(1.0)*4

        call rdimg( image)

        write(6,*) "enter number of degrees by which to rotate"
        read(5,*) theta

        image = image !gcon rotate( NROWS/2, NCOLS/2, theta*pi/180.0)

        call wrimg( image)
        end
```

```
      REAL TEMPLATE ROTATE( I, J, THETA)
C
C     When used in the context   A !gcon ROTATE( I, J, THETA)
C     this template rotates the image A THETA radians clockwise
C     about the point I, J
C     The method used causes some loss of information through
C     averaging in interpolation

      INTEGER Y1FLOOR, Y2FLOOR

      Y1DOT = Y1 - I
      Y2DOT = Y2 - J
      CT = COS(THETA)
      ST = SIN(THETA)
      Y1PRIME = Y1DOT*CT + Y2DOT*ST + I
      Y2PRIME = -Y1DOT*ST + Y2DOT*CT + J

      IF(Y1PRIME.GE.0.AND.Y1PRIME.LT.1) THEN
      Y1FLOOR = 0
      ELSE
      Y1FLOOR = NINT(SIGN(Y1PRIME, -1.0)/ABS(Y1PRIME)*0.5 + Y1PRIME)
      ENDIF

      IF(Y2PRIME.GE.0.AND.Y2PRIME.LT.1) THEN
      Y2FLOOR = 0
      ELSE
      Y2FLOOR = NINT(SIGN(Y2PRIME, -1.0)/ABS(Y2PRIME)*0.5 + Y2PRIME)
      ENDIF

      ROTATE( Y1FLOOR, Y2FLOOR+1 ) = ( 1 - ( Y1PRIME-Y1FLOOR ))*
     +      ( Y2PRIME-Y2FLOOR)
      ROTATE( Y1FLOOR+1, Y2FLOOR+1 ) = ( Y1PRIME-Y1FLOOR ) *
     +      ( Y2PRIME-Y2FLOOR )
      ROTATE( Y1FLOOR, Y2FLOOR ) = ( 1 - ( Y1PRIME-Y1FLOOR ))*
     +      ( 1 - ( Y2PRIME-Y2FLOOR ))
      ROTATE( Y1FLOOR + 1, Y2FLOOR ) = ( Y1PRIME-Y1FLOOR )*
     +      ( 1 - ( Y2PRIME-Y2FLOOR))
      END
```

Figure 18 shows the result of application of generalized convolution to an image and the rotation template.

Figure 18. Image **a** and image **a** rotated 23 degrees

SECTION V
INSTALLATION GUIDE

In this section, the procedure that must be followed to obtain an executable version of the IAF preprocessor that is tailored to the specific requirements of a particular installation is described. Several parameters in the preprocessor source code can be adjusted prior to its installation. These parameters primarily are used to establish the static size of various data structures within the preprocessor. In addition to the straightforward adjustments to particular parameters that can be effected, some slightly more involved modifications can be made to the source code of the preprocessor. For example, a combination of machine-specific and operating system-specific idiosyncrasies may necessitate a slight modification to the manner in which the preprocessor handles file I/O. It is helpful to understand, at least to some extent, the structure and operation of the IAF preprocessor to appreciate the modifications that one may make. Thus, a very brief overview of the preprocessor is given. Following that, the manner in which the preprocessor is customized and installed is outlined.

1. PREPROCESSOR OVERVIEW

The IAF preprocessor translates IAF source code into semantically-equivalent ANSI Standard FORTRAN 77 code. The preprocessor performs much of the sophisticated analysis of its input that is typically performed by a compiler. In particular, the preprocessor performs extensive lexical, syntactic, and semantic analysis of the IAF source code presented to it. In addition, the preprocessor has various code generation functions that are invoked to translate IAF code into its FORTRAN 77 equivalent.

The lexical analyzer is sufficiently general to recognize all of the lexical tokens of FORTRAN 77 (excluding `FORMAT` specifications) as well as the new tokens that were introduced by IAF. At the lexical level, some of the assumptions that the preprocessor must make are the maximum length of the input "card images" and the maximum number of characters that are considered significant in an identifier. Both of these limits can be modified in the source code of the preprocessor by adjusting appropriate parameters.

The preprocessor parses a significant portion of its input. Most of the FORTRAN 77 specification statements, as well as all of the executable statements that can potentially contain arithmetic expressions which take advantage of the Image Algebra extensions of IAF, are parsed. The parsing task requires a stack. In the unlikely event that the size of the

parse stack is exceeded during the translation of an IAF program, the maximum size of the parse stack can be increased and the preprocessor recompiled to reflect the increased stack size.

A symbol table is maintained by the preprocessor that stores assorted attributes of the identifiers that occur in an IAF program. Some of the information that is collected in the symbol table is gleaned from explicit specification statements and other information is inferred from the manner in which identifiers are used within executable statements. The semantic analysis of expressions that is performed by the preprocessor includes type checking and verification that the various program objects are used in conformance to the information stored in the symbol table and to the rules of FORTRAN 77 and IAF that govern their proper usage. In addition to the symbol table, a semantic stack is necessary to perform semantic analysis. The sizes of the symbol table and semantic stack can be adjusted in the preprocessor source code, if necessary.

Additional auxiliary data structures are required for performing code generation. In particular, there are stacks for handling actual arguments that arise in references to subprograms, a data structure adjunct to the symbol table that maintains information about arrays declared in the program, and other data structures for managing temporary scalar variables and temporary images that are generated by the preprocessor. The sizes of all of the arrays relevant to these data structures are also customizable.

In summary, the preprocessor is a tool of considerable sophistication, power, and flexibility. It can be readily and selectively reconfigured, on an individual basis, to accommodate the specific requirements of a given installation.

2. CUSTOMIZING THE PREPROCESSOR

Customizability of the IAF preprocessor is almost requited by the fact that the size of all FORTRAN 77 arrays that are local to a program unit are statically fixed at compile-time. A consequence of this property of FORTRAN 77 is that no matter how large the data structures internal to the IAF preprocessor are made, a moderately inventive programmer could, with little effort, compose a program that would exceed the size of one or more of the data structures. In any case, the preprocessor was designed to accommodate a diverse (and essentially unknown) user audience. Consequently, the source code of the preprocessor can be readily modified to suit the individual requirements of each recipient.

The vast majority of the modifications that can be made prior to compiling the preprocessor are trivial in nature and simply involve the adjustment of a FORTRAN 77 PARAMETER in the source code. A few others are only slightly more involved. A reasonable knowledge of FORTRAN 77 is desirable for carrying out these more advanced modifications. It is suggested that the sizes of the various data structures be left unchanged from what they are in the distributed source code and that the preprocessor be compiled for the first time with these default settings. Modifications to the preprocessor source can be made at future times as required by the mix of IAF source code that is commonly translated with it. Each time the preprocessor source code is modified, it must, of course, be subsequently recompiled in order to incorporate the modification(s) into a new executable image.

The source code of the IAF preprocessor contains comments of a particular form that are meant to be helpful towards locating the places in the code where the various modifications can be made. Each of these locator comments take the following general form.

```
CXXX-6LETID
```

The 'CXXX-' prefix is common to all of the locator comments. The '6LETID' suffix is a 6-letter mnemonic that is intended to be suggestive of the data structure and/or function that a particular comment marks.

a. Simple PARAMETERizations

The following code fragment appears in 64 places in the source code of the preprocessor.

```
CXXX-CDLEN
      INTEGER CDLEN
      PARAMETER (CDLEN=72)
```

The 'CDLEN' PARAMETER determines the maximum number of characters that are recognized in each line of input read by the preprocessor, as well as the length of the output lines that are produced by the preprocessor. An input line that contains more than 72 characters is effectively truncated; characters that occur beyond column 72 are ignored. Another reasonable setting for 'CDLEN' is 80.

The following code fragment occurs 12 times in the source code.

```
CXXX-PREFIX
      CHARACTER*2 PREFIX
      PARAMETER (PREFIX='XX')
```

The 'PREFIX' PARAMETER determines the first two characters that are used to prefix all of

the names that are generated by the preprocessor. The two characters 'XX' can be replaced with any two characters that can validly occur as a two-character prefix of a FORTRAN 77 identifier. However, note that modification of this PARAMETER effectively redefines the name space that is reserved for the preprocessor. In particular, the name space that is reserved for the preprocessor consists of all six-character identifiers that begin with the two characters contained in the 'PREFIX' PARAMETER.

The following fragment of code occurs in the FUNCTION called 'GETTOK' and the SUBROUTINE called 'YYCOPY'.

```
CXXX-MAXIDL
      INTEGER MAXIDL
      PARAMETER (MAXIDL=32)
```

The 'MAXIDL' PARAMETER determines the number of significant characters in an IAF identifier. Identifiers are allowed to exceed this length. However, all characters beyond character position 32 of an identifier are truncated.

The following fragment of code occurs in the FUNCTION called 'CLSSFY' and the FUNCTION called 'NEWLAB'

```
CXXX-LABRNG
      INTEGER LOWLAB, HGHLAB
      PARAMETER (LOWLAB=99000, HGHLAB=99999)
```

These two PARAMETERs determine the range of statement labels that are reserved for the preprocessor. If these two PARAMETERs are modified, 'LOWLAB' must be less than or equal to 'HGHLAB'. The preprocessor uses only labels that fall within the range delimited by these two PARAMETERs. The preprocessor comes to an abrupt halt when it runs out of statement labels. Therefore, if these default values are changed, the range of statement labels specified by the new values must be sufficient to satisfy the requirements of the preprocessor.

The following fragment of code occurs in the SUBROUTINE called 'ARRAYS'.

```
CXXX-ARRAYS
      INTEGER MAXARR
      PARAMETER (MAXARR=32)
```

The 'ARRAYS' SUBROUTINE manages the processing of array declarations and stores information relevant to the use of arrays as images. The 'MAXARR' PARAMETER determines the maximum number of arrays that can be declared in an IAF program unit. This includes arrays that are explicitly used as IAF images, as well as ordinary FORTRAN 77 arrays.

The following fragment of code occurs in the SUBROUTINE called 'BLIFST'.

```
CXXX-BIFSTK
      INTEGER STKSIZ
      PARAMETER (STKSIZ=64)
```

The 'BLIFST' SUBROUTINE aids in the translation of block IF statements. The 'STKSIZ' PARAMETER determines the maximum nesting allowed for block IF statements. A nesting depth of 64 should be far more than adequate.

The following fragment of code occurs in the SUBROUTINE called 'CALLST'.

```
CXXX-SUBREF
      INTEGER MAXARG
      PARAMETER (MAXARG=64)
```

The 'CALLST' SUBROUTINE translates CALL statements. The 'MAXARG' PARAMETER determines the maximum number of actual arguments that can be specified in a reference to a SUBROUTINE in a CALL statement.

The following fragment of code occurs in the SUBROUTINE called 'PUTCOM'.

```
CXXX-MAXTMP
      INTEGER MAXTMP
      PARAMETER (MAXTMP=256)
```

The 'MAXTMP' PARAMETER determines the maximum number of points in the source configuration of an IAF template. If the default value for the 'MAXTMP' PARAMETER is changed, then the run-time error handler (described below) must be modified accordingly.

The following fragment of code occurs in the SUBROUTINE called 'LEFTHS'.

```
CXXX-FNCREF
      INTEGER MAXARG
      PARAMETER (MAXARG=64)
```

The 'LEFTHS' SUBROUTINE translates FUNCTION references, arrays element references, and character substring references. The 'MAXARG' PARAMETER determines the maximum number of actual arguments that can occur in a reference to a FUNCTION within an expression.

The following fragment of code occurs in the FUNCTION called 'PARSTK'.

```
CXXX-PARSTK
      INTEGER STKSIZ
      PARAMETER (STKSIZ=128)
```

The 'PARSTK' FUNCTION manages the parse stack. The 'STKSIZ' PARAMETER determines

the maximum size of the parse stack.

The following fragment of code occurs in the SUBROUTINE called 'SCALAR'.

```
CXXX-SCALAR
      INTEGER MAXSCL
      PARAMETER (MAXSCL=32)
```

The 'SCALAR' SUBROUTINE manages the allocation of temporary scalar variables that are generated by the preprocessor for storing intermediate scalar results that arise during the evaluation of expressions. The 'MAXSCL' PARAMETER determines the maximum number of temporary scalar variables that can be used for evaluating all of the expressions that occur in a single IAF statement.

The following fragment of code occurs in the SUBROUTINE called 'SEMSTK'.

```
CXXX-SEMSTK
      INTEGER STKSIZ
      PARAMETER (STKSIZ=64)
```

The 'SEMSTK' SUBROUTINE manages the semantic stack. The 'STKSIZ' PARAMETER determines the maximum size of the semantic stack.

The following fragment of code occurs in the SUBROUTINE called 'STATIC'.

```
CXXX-STATIC
      INTEGER MAXSTC
      PARAMETER (MAXSTC=32)
```

The 'STATIC' SUBROUTINE manages the allocation of temporary images that are generated by the preprocessor for storing intermediate image results that arise during the evaluation of expressions. The 'MAXSTC' PARAMETER determines the maximum number of temporary images that can be used for evaluating all of the expressions that occur in a single IAF statement.

The following fragment of code occurs in the SUBROUTINE called 'STRNGS'.

```
CXXX-STRING
      INTEGER MAXSTR, AVGLEN
      PARAMETER (MAXSTR=64, AVGLEN=32)
```

The 'STRNGS' SUBROUTINE manages a table of character strings that is used to temporarily store miscellaneous fragments of code text that are produced by the preprocessor during code generation. The 'MAXSTR' PARAMETER determines the maximum number of strings that can be stored in the string table during the translation if any given IAF statement. The 'AVGLEN' PARAMETER is an estimate of the average length of a string. The total number of

characters that the string table can accommodate is given by the expression 'MAXSTR*AVGLEN'. Note that this character total can be exceeded with fewer than 64 strings if their average length is greater than 32. Thus, strings are entered into the string table until either the total number of strings exceeds 64 or the total number of characters exceeds 2048, whichever occurs first.

The following fragment of code occurs in the FUNCTION called 'SYMTBL'.

```
CXXX-SYMTBL
      INTEGER MAXSYM, AVGLEN
      PARAMETER (MAXSYM=500, AVGLEN=8)
```

The 'SYMTBL' FUNCTION manages the symbol table of the IAF preprocessor. The 'MAXSYM' PARAMETER determines the maximum number of identifiers that can be stored in the symbol table during the translation of any given IAF program unit. The 'AVGLEN' PARAMETER is an estimate of the average length of an identifier. The total number of characters that the symbol table can accommodate is given by the expression 'MAXSYM*AVGLEN'. Note that this character total can be exceeded with fewer than 500 identifiers if the average length of the identifiers is greater than 8. Thus, identifiers are entered into the symbol until either the total number of identifiers exceeds 500 or the total number of characters exceeds 4000, whichever occurs first. The names of the ANSI Standard FORTRAN 77 INTRINSIC functions are *always* kept in the symbol table.

b. File I/O

Portions of the source code of the preprocessor that help handle file input and output functions are tagged with locator comments, as well. If the management of I/O has to be slightly adjusted to conform to a particular machine or operating system, these comments direct one to the relevant locations in the source code.

The preprocessor was developed under the UNIX operating system and is expected to readily port to any UNIX environment. Very little difficulty was encountered in porting the preprocessor to a VMS environment. When ported to a VMS-based machine, several FORMAT specifications may need to be replaced with the alternatives noted below. Given that the preprocessor was implemented in strict conformance to ANSI Standard FORTRAN 77, it is expected to be easily portable, in general. Only time and experience can tell for sure.

The following fragment of code occurs in the SUBROUTINE called 'WRITCD' and the SUBROUTINE called 'ERRORS'.

```
CXXX-STDERR
      INTEGER STDERR
      PARAMETER (STDERR=0)
```

The 'STDERR' PARAMETER determines the unit identifier of the destination of diagnostic error messages that are produced by the IAF preprocessor.

The following fragment of code occurs in the SUBROUTINE called 'WRITCD'.

```
CXXX-FORMAT
      WRITE (UNIT=C2, FMT='(I2)') CDLEN
      IOFMTS(1) = '(A'//C2//')'
C IF VMS THEN REPLACE PRECEDING LINE WITH THE FOLLOWING LINE.
C     IOFMTS(1) = '(1X,A'//C2//')'
      WRITE (UNIT=C2, FMT='(I2)') CDLEN-1
      IOFMTS(2) = '(''*'',A'//C2//')'
C IF VMS THEN REPLACE PRECEDING LINE WITH THE FOLLOWING LINE.
C     IOFMTS(2) = '(1X,''*'',A'//C2//')'
```

The FORMAT specifications that are set up in this piece of code are used during reading from the standard input and writing to the standard output. The embedded comments indicate alternative FORMAT specifications that are suggested for VMS environments.

The following fragment of code occurs in the FUNCTION called 'READCD'.

```
CXXX-FILEIO
      READ (UNIT=*, FMT=IOFMTS(1), ERR=7000, END=8000) CARD
```

This is the READ statement that effects read actions on the standard input file.

The following fragment of code occurs in the SUBROUTINE called 'WRITCD'.

```
CXXX-FILEIO
      FDPERM = 6
      FDTEMP = 2
      OPEN (UNIT=FDTEMP, STATUS='scratch', ACCESS='direct',
     $FORM='formatted', RECL=CDLEN, ERR=7100)
```

The 'FDPERM' PARAMETER determines the unit identifier of the standard output file. The 'FDTEMP' PARAMETER determines the unit identifier of a temporary file that is used during translation. This temporary file collects code that is generated by the preprocessor and is intermittently appended to the standard file. Random access to this temporary file is required.

The following fragment of code occurs in the SUBROUTINE called 'WRITCD'.

```
CXXX-FILEIO
        WRITE (UNIT=FDPERM, FMT=IOFMTS(1), ERR=7200) SCARD
      ELSE
        WRITE (UNIT=FDTEMP, FMT=IOFMTS(1), REC=NXTREC, ERR=7200) SCARD
```

These are the WRITE statements that write to the standard output file and the temporary file. Any statement code produced by the preprocessor is written by one of these statements.

The following fragment of code occurs in the SUBROUTINE called 'WRITCD'.

```
CXXX-FILEIO
        WRITE (UNIT=FDPERM, FMT=IOFMTS(2), ERR=7200) SCARD
      ELSE
        WRITE (UNIT=FDTEMP, FMT=IOFMTS(2), REC=NXTREC, ERR=7200) SCARD
```

These WRITE statements also write on the standard output file and the temporary file. These write statements emit comment lines.

The following fragment of code occurs in the SUBROUTINE called 'WRITCD'.

```
CXXX-FILEIO
        READ (UNIT=FDTEMP, FMT=IOFMTS(1), REC=I, ERR=7300) CARD
        WRITE (UNIT=FDPERM, FMT=IOFMTS(1), ERR=7200) CARD
```

This pair of statements concatenates the temporary file to the tail of the standard output file.

c. Minimum and Maximum Representable Values

The following fragment of code occurs in the SUBROUTINE called 'IMGTMP'.

```
CXXX-LIMSET
      LOGICAL LIMSET
      PARAMETER (LIMSET=.FALSE.)
```

The LOGICAL value to which the 'LIMSET' PARAMETER is set determines a code-generation option affecting the '!amax', '!amin', '!mmax', and '!mmin' image-template operations. In particular, if the value of 'LIMSET' is set to '.FALSE.' (the default), then the preprocessor will generate code to effect the above image-template operations that initializes the computation on a particular target image pixel based on the first source weight configuration that is referenced in the respective template. On the other hand, if the value of 'LIMSET' is '.TRUE.', then the preprocessor will generate code for the above image-template operations that initializes the computation on a given target pixel based on the minimum and maximum values that representable by the processor on which the IAF programs are developed.

The following fragment of code occurs in the SUBROUTINE called 'PUTLIM'.

```
CXXX-LIMITS
      INTEGER MXILEN
      PARAMETER (MXILEN=10)
      CHARACTER*(MXILEN) MXISTR
      PARAMETER (MXISTR='2147483647')
      INTEGER MNILEN
      PARAMETER (MNILEN=11)
      CHARACTER*(MNILEN) MNISTR
      PARAMETER (MNISTR='-2147483648')
      INTEGER MXRLEN
      PARAMETER (MXRLEN=11)
      CHARACTER*(MXRLEN) MXRSTR
      PARAMETER (MXRSTR='3.40282E+38')
      INTEGER MNRLEN
      PARAMETER (MNRLEN=12)
      CHARACTER*(MNRLEN) MNRSTR
      PARAMETER (MNRSTR='-3.40282E+38')
      INTEGER MXDLEN
      PARAMETER (MXDLEN=20)
      CHARACTER*(MXDLEN) MXDSTR
      PARAMETER (MXDSTR='1.7976931348623D+308')
      INTEGER MNDLEN
      PARAMETER (MNDLEN=21)
      CHARACTER*(MNDLEN) MNDSTR
      PARAMETER (MNDSTR='-1.7976931348623D+308')
```

This code is relevant only if the 'LIMSET' PARAMETER is set to '.TRUE.'. The numbers reflected as character strings in the above code correspond to the minimum and maximum representable integer and floating point values for Sun Workstations.

This code can be tailored to apply to a particular processor. The various PARAMETERs in the above code that have 'MN' (resp. 'MX') as a prefix correspond to minimum (resp. maximum) values. The third character in the names corresponds the data type of the values where 'I' denotes INTEGER, 'R' denotes REAL, and 'D' denotes DOUBLE PRECISION. For minimum representable values, the 'MN?STR' PARAMETER should be defined to the character string that denotes the respective minimum representable value and the 'MN?LEN' PARAMETER set to the number of characters in that string. The PARAMETERs that correspond to maximum representable values, 'MX?LEN' and 'MX?STR', should be similarly defined.

d. FORTRAN 77 INTRINSIC Functions

The following fragment of code occurs in the SUBROUTINE called 'STINIT'.

```
CXXX-INTRIN
      DATA INTNAM(1),NAMLEN(1),INTTYP(1) /'INT    ',3,TYINT4/
      DATA INTNAM(2),NAMLEN(2),INTTYP(2) /'IFIX   ',4,TYINT4/
      DATA INTNAM(3),NAMLEN(3),INTTYP(3) /'IDINT  ',5,TYINT4/
      DATA INTNAM(4),NAMLEN(4),INTTYP(4) /'REAL   ',4,TYREAL/
      DATA INTNAM(5),NAMLEN(5),INTTYP(5) /'FLOAT  ',5,TYREAL/
      DATA INTNAM(6),NAMLEN(6),INTTYP(6) /'SNGL   ',4,TYREAL/
      DATA INTNAM(7),NAMLEN(7),INTTYP(7) /'DBLE   ',4,TYDBLE/
      DATA INTNAM(8),NAMLEN(8),INTTYP(8) /'CMPLX  ',5,TYCMPL/
      DATA INTNAM(9),NAMLEN(9),INTTYP(9) /'ICHAR  ',5,TYINT4/
      DATA INTNAM(10),NAMLEN(10),INTTYP(10) /'CHAR   ',4,TYCHAR/
      DATA INTNAM(11),NAMLEN(11),INTTYP(11) /'AINT   ',4,TYGNRC/
      [. . .]
      DATA INTNAM(80),NAMLEN(80),INTTYP(80) /'TANH   ',4,TYGNRC/
      DATA INTNAM(81),NAMLEN(81),INTTYP(81) /'DTANH  ',5,TYDBLE/
      DATA INTNAM(82),NAMLEN(82),INTTYP(82) /'LGE    ',3,TYLOG/
      DATA INTNAM(83),NAMLEN(83),INTTYP(83) /'LGT    ',3,TYLOG/
      DATA INTNAM(84),NAMLEN(84),INTTYP(84) /'LLE    ',3,TYLOG/
      DATA INTNAM(85),NAMLEN(85),INTTYP(85) /'LLT    ',3,TYLOG/
CXXX-INTRIN
      INTEGER NUMINT
      PARAMETER (NUMINT=85)
```

Before the translation of each IAF program unit, the symbol table of the preprocessor is initialized with the names of the FORTRAN 77 INTRINSIC functions. The above code is relevant to that initialization. In all, there are 85 INTRINSIC functions that are listed in ANSI X3.9-1978. This total includes both specifically-typed and generically-typed INTRINSIC functions.

This list can be extended as appropriate to include any additional INTRINSIC functions that have been added to a particular FORTRAN 77 implementation. To effect this modification, the 'NUMINT' PARAMETER is adjusted to reflect the new total number of INTRINSIC functions. In addition, one DATA statement similar to the ones shown above is added for each new INTRINSIC function that is introduced. The specification of an INTRINSIC function contains three constituents (i) the name of the function as a character string, (ii) the number of characters in the name, and (iii) the type of the result returned by the function. The 'INTNAM(1)' array element holds the name of the function where it is assumed that the name has at most six characters. The 'NAMLEN(1)' array element is set to the number of characters in that name. The 'INTTYP(1)' array element is set to the type that is returned by the so named FUNCTION. The types are specified using PARAMETERs

that are suitably defined in the source code of the preprocessor. Specifically, the **PARAMETER** named '**TYINT4**' (resp. '**TYREAL**', '**TYDBLE**', '**TYCMPL**', '**TYCHAR**', '**TYLOG**') corresponds to type **INTEGER** (resp. **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **CHARACTER**, **LOGICAL**). The **PARAMETER** '**TYGNRC**' is used to denote that a **FUNCTION** is generically-typed.

e. Special Note for VMS Sites

In addition to the modifications to **FORMAT** specifications that were mentioned above, a couple of other minor issues are relevant to VMS installations. In particular, it may be desirable to remove the **STOP** statement from the main program unit of the preprocessor. Otherwise, a "**FORTRAN STOP**" message is issued whenever the preprocessor terminates normally. In addition, the following DCL commands are suggested to help in presenting a nice user interface to the IAF preprocessor.

```
ASSIGN infile SYS$INPUT
ASSIGN outfile SYS$OUTPUT
```

Here, '**infile**' is the name of the file containing IAF source code to be preprocessed and '**outfile**' is the name of the file that the preprocessed output is to be written to.

3. RUN-TIME ERROR-HANDLER

The IAF preprocessor generates code for reporting two detected run-time errors. The code that is generated includes a **CALL** to a generic run-time error-handling **SUBROUTINE** that is invoked in the event that one of these execution-time errors occurs. The following is the run-time error handling **SUBROUTINE** that is assumed by the preprocessor.

```
      SUBROUTINE XXERRH(LINENO, ERRNO)
      INTEGER LINENO, ERRNO
      INTEGER STDERR, MAXTMP
      PARAMETER (STDERR=0, MAXTMP=256)
      INTEGER NONCNF, TMPOVF
      PARAMETER (NONCNF=1, TMPOVF=2)
      WRITE(UNIT=STDERR,FMT=9000) LINENO
      IF (ERRNO.EQ.NONCNF) THEN
        WRITE(UNIT=STDERR,FMT=9010)
      ELSEIF (ERRNO.EQ.TMPOVF) THEN
        WRITE(UNIT=STDERR,FMT=9020) MAXTMP
      ENDIF
 9000 FORMAT('*** Abnormal termination at "iaf" source line ', I4)
 9010 FORMAT('*** Image coordinate-set mismatch.')
 9020 FORMAT('*** Maximum template size of ', I4, ' exceeded.')
      STOP
      END
```

The two run-time errors that are detected are (i) the attempt to perform some operation involving two or more images in which at least two of the images have distinct coordinate sets (and are, therefore, incompatible for the specified operation), and (ii) an overflow of the maximum number of source configuration points that are allowed in an IAF template. If one of these errors does occur at run-time, an appropriate error message is emitted by the error handler and execution of the IAF program is terminated. The diagnostic will include the number of the IAF source line that contains the offending code.

If a copy of this SUBROUTINE was not provided on the distribution tape, then it should be manually typed in and made available during the linking phase of the compilation of an IAF program. The 'STDERR' PARAMETER determines the unit identifier to which error messages are to be directed. The 'MAXTMP' PARAMETER determines the maximum number of source configuration points that are allowed in a template. These two PARAMETERs should be adjusted appropriately to conform to the values that they are declared to have in the source code of the preprocessor.

4. COMPILING THE PREPROCESSOR

Once the source code of the IAF preprocessor has been customized as required, it can be compiled and installed. All that is required to accomplish this task is a FORTRAN 77 compiler. The preprocessor is implemented in fairly strict conformance to ANSI Standard FORTRAN 77. Hence, provided that the on-site customizations were properly done, one should have no difficulty in compiling the preprocessor. The preprocessor was developed under the

UNIX operating system and ported with ease to a machine operating under VMS. With respect to the '`f77`' compiler of many UNIX systems, it is generally necessary to increase the size of the symbol table that is used by the compiler for storing external identifiers. The size of this table is increased by supplying the '`-Nx`*nnn*' flag to '`f77`' where '*nnn*' indicates the size of the table that is desired. The default size of the symbol table is 900. A size of 2000 is more than adequate for compiling the preprocessor. After compilation, the executable image of the preprocessor that was created should be appropriately integrated into the set of user utilities that is maintained on the machine and/or an appropriate user-interface to the preprocessor should be established by the system manager.

As stated earlier, the IAF preprocessor does not subvert the separate compilation features of FORTRAN 77. In that regard, libraries of IAF subprograms and templates can be generated by (i) translating the relevant source code through the IAF preprocessor, (ii) compiling the preprocessed output with a FORTRAN 77 compiler, and (iii) archiving the resulting object code. One word of caution about the use of archived libraries is in order. Whenever the preprocessor is recompiled, the libraries should be regenerated using the updated version of the preprocessor. Depending on the changes made to the preprocessor, code translated with an obsolete version of the preprocessor may be incompatible with code translated with the new version. To be safe, it is recommended that libraries be regenerated in-full whenever the preprocessor is modified. Otherwise, IAF programs may exhibit unpredictable behavior at run-time.

5. ANSI NONCOMPLIANCE

The source code of the preprocessor is known to be in violation of the ANSI Standard for FORTRAN 77 in at least one respect. In particular, liberal use was made of the full set of printable ASCII characters which, of course, is a proper superset of the ANSI Standard FORTRAN 77 character set. Although the IAF source code is mapped to upper-case before it is distributed, lower-case letters within character constants are left untouched by the case-mapping utility. Lower-case letters are primarily contained within error messages that are produced by the preprocessor. If it is preferred to have the user presented with error messages that are all in upper-case, then the character constants in the source code of the preprocessor must be massaged appropriately. In any case, the FORTRAN 77 character set had to be abrogated in order to accommodate certain lexical tokens that have been introduced to IAF (e.g., '`@`', '`!max`', etc.). No apologies are given, nor should they be expected, for this justifiable infringement of the ANSI Standard.

APPENDIX A
IAF ARITHMETIC EXPRESSION GRAMMAR

This appendix contains a BNF grammar that describes the syntax of IAF arithmetic expressions. The grammar rules listed below are used by the preprocessor to parse IAF arithmetic expressions. Entities enclosed within single quotes (e.g., '*', '!dot', '==', etc.) are literal strings that can appear in expressions. The capitalized entity IDENT (resp., ICONS, RCONS, DCONS, CCONS) represents an arbitrary element in the set of valid FORTRAN 77 identifiers (resp., **INTEGER** constants, **REAL** constants, **DOUBLE PRECISION** constants, **COMPLEX** constants). The grammar was designed in such a way as to impose the proper precedence upon the IAF arithmetic operators. Aside from the following two exceptions, all binary arithmetic operators are left-associative.

(1) The power operator ('**') is right-associative.

(2) The characteristic-function operators ('<', '<=', '==', '!=', '>', '>=') are non-associative.

The intent of this grammar is solely to describe the syntactically valid IAF arithmetic expressions. The grammar does not enforce semantic constraints imposed on arithmetic expressions by the definitions of FORTRAN 77 and IAF. For example, the syntactic category "arg_list" is used to generate the actual argument lists of arithmetic function references and the index expression lists of arithmetic array element references. The syntactic categories "char_expr", "rel_expr", and "log_expr" (the BNF productions for which are not reproduced here) that are used to generate character expressions, relational expressions, and logical expressions, respectively, are syntactically valid within the "arg_list" of a function reference and an array element reference. However, they are semantically valid only within the "arg_list" of a function reference. The semantic correctness of arithmetic expressions is verified and enforced by various preprocessor functions that are invoked during the parsing of expressions.

arith_expr → add_expr char_func_op add_expr | add_expr

char_func_op → '<' | '<=' | '==' | '!=' | '>' | '>='

add_expr → add_expr add_op bibang_expr | uni_op bibang_expr | bibang_expr

add_op → '+' | '-'

uni_op → add_op | '!inv' | '!max' | '!min' | '!sum'

bibang_expr → bibang_expr bi_bang_op term | term

bibang_op → '!dot' | '!max' | '!min'

term → term mult_op image_temp_expr | image_temp_expr

mult_op → '*' | '/'

image_temp_expr → image_temp_expr image_temp_op var_ref | factor

image_temp_op → '!amax' | '!amin' | '!gcon' | '!mmax' | '!mmin'

factor → primary '**' factor | primary

primary → var_ref | constant | '(' arith_expr ')'

var_ref → IDENT | IDENT '()' | IDENT '(' arg_list ')'

arg_list → arg_list ',' argument | argument

argument → arith_expr | '@' arith_expr | char_expr | rel_expr | log_expr

constant → ICONS | RCONS | DCONS | CCONS

APPENDIX B
PREPROCESSOR-RESERVED NAMES

All names that are six characters long and which begin with 'XX' are reserved for the preprocessor. The use of a reserved name as an identifier within an IAF program is prohibited. If the preprocessor encounters an identifier that encroaches on the reserved name space within an IAF program, a diagnostic warning message is issued informing the user of the infraction. The behavior of the preprocessor on IAF source code that utilizes one or more reserved names as identifiers is undefined.

The preprocessor actually uses a small subset of the reserved names to identify entities generated by the preprocessor. The names explicitly used by the IAF preprocessor are listed below along with a brief description of their role. In the following, '*d*' represents an arbitrary decimal digit.

XXSI*dd* INTEGER variables that are generated by the preprocessor for storing the results of expressions that produce INTEGER scalars.

XXSR*dd* The REAL analog of XXSI*dd*.

XXSD*dd* The DOUBLE PRECISION analog of XXSI*dd*.

XXSC*dd* The COMPLEX analog of XXSI*dd*.

XXSL*dd* The LOGICAL analog of XXSI*dd*.

XXSS*dd* The INTEGER*2 analog of XXSI*dd*.

XXI*ddd* INTEGER arrays that are generated by the preprocessor for the purpose of storing the results of expressions that produce INTEGER images.

XXR*ddd* The REAL analog of XXI*ddd*.

XXD*ddd* The DOUBLE PRECISION analog of XXI*ddd*.

XXC*ddd* The COMPLEX analog of XXI*ddd*.

XXS*ddd* The INTEGER*2 analog of XXI*ddd*.

XXB*ddd* One XXB*ddd* INTEGER array is created by the preprocessor for each statement that contains one or more IAF expressions which require preprocessor-generated temporary images to evaluate them. This array is made large enough to fit all of the temporary images created by the preprocessor for storing the intermediate results which arise during the evaluation of the expression(s).

XXO*ddd* INTEGER PARAMETERs set to the values of appropriate offsets into the XXB*ddd* arrays at which preprocessor-generated temporary images are EQUIVALENCEd.

XXZ*ddd* INTEGER PARAMETERs set to the sizes of preprocessor-generated temporary images.

XXA*ddd* INTEGER variables that store the values of the dimension bounds of programmer-defined images.

XXLCV1 An INTEGER variable used as the control variable of a DO-loop that iterates over the x-coordinate of a programmer-defined image.

XXLCV2 An INTEGER variable used as the control variable of a DO-loop that iterates over the y-coordinate of a programmer-defined image.

XXIND*d* INTEGER variables used to iterate over the pixels of preprocessor-generated temporary images.

XXITWT An INTEGER array that stores the weight configuration of a template that has INTEGER weights.

XXRTWT The REAL analog of XXITWT.

XXDTWT The DOUBLE PRECISION analog of XXITWT.

XXCTWT The COMPLEX analog of XXITWT.

XXMAXT An INTEGER PARAMETER that determines the maximum number of points allowed in the source configuration of each programmer-defined template. The value of this PARAMETER becomes fixed at the time when the preprocessor is installed and can be adjusted prior to its compilation. The value of XXMAXT is initially set to 256.

XXTHWM An INTEGER variable that stores the value of the number of points in the source configuration of a template. The value of XXTHWM becomes defined during an image-template operation whereby it is set to the number of points in the source configuration of the template referenced in the operation.

XXOFY1 An INTEGER array used during an image-template operation to store the x-offsets, from the target image pixel, of the points in the source configuration of the template.

XXOFY2 The analog to XXOFY1 that is relevant to the y-offset.

XXTIY1 An INTEGER variable that indexes the x-coordinate of a target pixel during an image-template operation.

XXTIY2 The analog to XXTIY1 that is relevant to the y-coordinate.

XXTLCV An INTEGER variable used as the control variable of a DO-loop that iterates over the points in the source configuration of a template.

XXTFLG A LOGICAL variable that flags when a pixel in the target image of an image-template operation is set to a known value. Note that XXTFLG is not relevant to the !gcon image-template operation.

XXCOM*d* The names of COMMON blocks used to implement image-template operations. They effect the sharing of relevant data between program units which contain image-template operations and the corresponding template definitions. For example, the arrays which store the source and weight configurations of templates are placed in these COMMON blocks.

XXERRH The name of a run-time error-handling SUBROUTINE.

In addition to the reserved names listed above, the INTEGER variables 'Y1' and 'Y2' have special meaning within template definitions. The discussion of templates described the role of these names in template definitions. Use of these variables in a template definition in a manner which contravenes that which is prescribed is prohibited.

REFERENCES

1. G.X. Ritter, et. al., "Standard Image Processing Algebra Document Phase II." TR (7) Image Algebra Project, F08635-84-C-0295, Eglin AFB, FL (1987).